The Pragmatic Programmers

TURING 图灵程序设计丛书

Ship it! A Practical Guide to Successful Software Projects

软件项目成功之道



[美] Jared R. Richardson
William A. Gwaltney Jr. 著
苏金国 王少轩 等译

人民邮电出版社
POSTS & TELECOM PRESS

Jared R. Richardson

开发人员、演说家、作家和独立顾问。他由开发人员逐步成长为研发经理，有着十多年丰富的工作经验，擅长使用非定制技术来解决疑难问题。他领导着SAS软件研究所的一个开发和测试团队，带领整个公司提高了测试自动化的效率。

William A. Gwaltney Jr.

有着二十多年软件开发经验，在网络、通信及基于网络的计划调度等方面都很有造诣。他在SAS软件研究所从事测试自动化方面的工作。

图灵程序设计丛书

Ship It!
A Practical Guide to Successful Software Projects
软件项目成功之道



[美] Jared R. Richardson
William A. Gwaltney Jr. 著
苏金国 王少轩 等译

人民邮电出版社
北京

图书在版编目（CIP）数据

软件项目成功之道 / (美) 理查森
(Richardson,J.R.)，(美) 格沃特尼 (Gwaltney,W.A.)
著；苏金国等译. -- 北京：人民邮电出版社，2011.8
(图灵程序设计丛书)
书名原文：Ship It! A Practical Guide to
Successful Software Projects
ISBN 978-7-115-25965-3

Ⅰ. ①软… Ⅱ. ①理… ②格… ③苏… Ⅲ. ①软件开
发—项目管理 Ⅳ. ①TP311.52

中国版本图书馆CIP数据核字(2011)第141577号

内 容 提 要

作者以精炼、风趣的语言揭开了项目管理过程的神秘面纱。所涵盖的内容涉及工具、使用项目技术、曳光弹开发以及常见问题的解决办法，并提供大量实用建议，且总结出方方面面的“技巧”，帮助读者在阅读过程中快速消化所看内容。

本书适合软件研发专业人士阅读，对软件项目管理感兴趣的社会各界人士也能从中获益。

图灵程序设计丛书

软件项目成功之道

◆ 著　[美] Jared R. Richardson　William A. Gwaltney Jr.
译　苏金国　王少轩 等
责任编辑　傅志红　卢秀丽

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：700×1000　1/16
印张：12.25
字数：213千字　2011年8月第1版
印数：1－4 000册　2011年8月北京第1次印刷
著作权合同登记号　图字：01-2011-1378号
ISBN 978-7-115-25965-3

定价：39.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

译 者 序

说来也巧，拿到原书正在翻看时，我所在的一个项目组临时召集开会。这个项目去年就已经启动，甚至单位最高领导亲自动员，其重要性可见一斑。开始时大家确实士气高昂，成效明显，每周例会都会汇报新的进展，可惜第一个演示版本提交后，这种势头就没能再保持下去。大家都忙于处理自己手上其他的繁杂事务，在这个项目上投入的精力越来越少，每周例会也不再召开，已经成为回忆。由于中期检查临近，这才不得不召开“例会”了解目前工作状态，并讨论下一步工作安排。

会议开始后，一个同事的发言让我觉得似曾相识。他负责的部分需要与外单位开发的 4 个模块集成，提交演示版本时，只有一个模块能成功集成，另外 3 个模块一直有问题。这期间改改停停，问题一直不能干净利落地得到解决。最近联系过一次联调，经过几天努力，终于搞定了这 3 个模块。可是最后一天外单位人员临走时过于匆忙，不小心覆盖了那几天的成果。所以这位同事说，只好再安排一个时间重新联调，不过时间还没有定下来……我心里想：这不正是我刚才在书里偶然看到的情况吗？

这时另一位同事也表示出对这个单位合作态度的不满，开始大谈他们的种种不是，大家只能百无聊赖地听着，不好意思打断。接下来话题扯得越来越远，一下午的会议结束后，尽管大家都很是辛苦，甚至头昏脑胀，但还是不清楚目前到底有哪些新的进展，也没有讨论好接下来的具体安排。

回到自己的办公室，重新拿起这本书，我发现这一切正是书中所说的一个真实场景的再现，当即把它推荐给了我们的项目负责人。现如今，我们的每周例会已经恢复（尽管没有做到书中建议的每日例会），项目在有条不紊地向前推进。

关于这本书的内容，我不想多讲，只想说：这本小书会带给你实实在在的

好处，如果你手上有迟迟无法交付的项目，也许这就是你的曙光！

“好风凭借力，送我上青云”！借着大师们的力量，站在巨人的肩上，我们会走得更远！

全书主要由苏金国、王少轩翻译，并得到王小振、李璜、刘亮、李新宏的大力协助。若译文有不当之处，敬请读者批评指正。

读者对本书的赞誉

真是让人难以置信的一本书。刚开始读到这本书，我惊讶得几乎要从座位上掉下来，因为那时我参与的一个项目正在遭遇同样的问题，简直与你描述的一模一样，这本书会让我获益多多。

► Matthew Bass，软件工程师

就像 Mac 一样，这本书“相当实用”，因为它从团队领导人和成员的经验教训中汲取了精华，揭开了软件项目管理过程的神秘面纱……这本书语言非常简洁，很快就能读完，如果你是软件开发行业中的一员，花几个小时读读这本书绝对值得。

► Robert Pritchett，macCompanion 公司

在软件行业，如果我们都不再像晕头转向的小丑那样整天忙得团团转，而是能够充分发挥自己的潜能，那就太好了。这本书可以提供帮助，只要那些真正需要帮助的人能够关注本书。

► Mike Gunderloy，Larkware 公司

如果你交给别人一本书，而它居然能改变他们的思维和行为方式，这种感觉确实很奇妙。我真的很高兴读到这样一本书，相信以后还会不时拿来翻看，这本书就是《软件项目成功之道》。

► Jeffery Fredrick，CruiseControl 平台

很少有一本书能让开发人员和经理们都信服，不过这本书确实做到了……这本书中我最喜欢的是最后一部分，汇集了一些篇幅不长的文章，不仅描述了软件项目经常遇到的问题，还指出了如何应用书中给出的原则和实践解决这些问题。与我读过的另外一些牵强附会的“反模式”总汇不同，这部分非常实用，

确实很有用。

▶ Ernest Friedman-Hill，Java Ranch

真是一本好书！作者用通俗自然的笔调将内容娓娓道来，没有落入刻板介绍方法的俗套，实在太棒了。

▶ Roberto Gianassi，IT 咨询师

《程序员修炼之道》为我们介绍了软件开发人员个人的技术和能力，这本书则是面向软件开发团队。如果你认识的同在软件领域打拼的某个人工作上不顺心，可以让他读一读这本书。

▶ Guerry A. Semones，资深软件工程师，Appistry 公司

一本关于软件的书居然如此有趣，实属难得。这些想法睿智、中肯而且非常重要。此时此刻，我读到的这些内容会让我立刻成为一个更棒的程序员。

▶ Joe Fair

这本书保持了 Pragmatic 图书一贯的风格，读起来轻松流畅，简明扼要。我只用两天就读完了这本书，收获颇丰，而且从中掌握的精髓要义可以直接加以应用。如果你想让自己的软件开发生涯更上一个台阶，强烈推荐这本书！

▶ Anil John

如果你是一个开发团队领导人或者开发经理，不读这本书会被“炒鱿鱼”的。

▶ David Starr，Elegant Code 公司

教条并不意味着没有思想，而是思想的终结。

▶吉尔伯特•基思•切斯特顿（1874—1936）

序　言

你可能已经注意到了，书架上关于软件开发的书并不只有这一本。

实际上，关于设计和构建软件的书林林总总，多得让人眼花缭乱，不仅如此，它们相互之间还不太一致。遗憾的是，这些方法上的分歧带来的感受往往让人“炙热难当”：作为软件行业的从业人员，我们体会到的不是光明，而是焦头烂额的感觉。另外，我们的项目总是延期。

我们在不断探寻更好的软件开发方法，希望能找到适合自己和团队的好办法。不过，基于既有的教条，关于各种开发方法孰优孰劣的讨论最终总会演变成激烈的争吵。字典中教条的定义是“一种权威性观点，但并没有充分的依据”。我们经常会看到，各种方法的拥护者们都坚持认为自己的方法才是开发软件唯一正确的方法。我们不断听到一些从业人员这么讲，他们执着地采用某种方式开发软件，即使这种方法明显危害到团队的其他人甚至整个组织，却仍然固执己见。

事实上，开发软件根本没有所谓“绝对正确的方法”。倒是有很多错误的方法，不过没有哪一种方法、观点、哲学或工具能“以不变应万变”，在所有时间、所有场合对所有项目和所有人都适用。软件是人创建的，不会有两个人完全一样。

所以对于软件开发实践，我们还是沿用“务实”的观点。我们要强调的是目标：你希望得到一堆签名，还是希望所有人都能理解？你希望匆忙间随便抛出一堆东西来，还是生成一个可以真正帮助别人完成工作的软件？

我们想尝试新鲜事物，并对我们的实践不断评价和修正；我们希望找到适用的方法。不过着手这个工作很费功夫：我们要做大量研究，需要耗费大量时

间，而大多数参与实际工作的程序员根本没有这么多时间。

这让我萌生出一个想法：请 Jared 和 Will 写这样一本书。这是一本快速入门的指南，你能从中了解开发可靠代码所需的基本、有效的工具和技术。

Jared 和 Will 是《程序员修炼之道》最早的读者，他们用心领会书中的道理，运用我们的方法和技术，并结合其他流行的敏捷方法实践，形成了一种独特的方法，让他们不论在小的初创公司还是世界上最大的私营软件公司，都能得心应手地开展工作。

这本书汇集了他们最钟爱的技术和实践。运用这些内容，你将在改善开发过程时如虎添翼，快速入手，当然可能还需要补充 Pragmatic Starter Kit 系列介绍的其他一些技术细节。随着时间的推移，你可能希望进一步扩展，尝试另外一些实践和技术。毕竟，这正是务实之道——根据当务之急，找到最适用的方法。

希望这本书对你有帮助，能让你轻轻松松地成功交付系统！

Andy Hunt
The Pragmatic Programmers 公司
2005 年 4 月
andy@pragmaticprogrammer.com

前　　言

不论是对你自己还是对你的职业发展，最明智的一项投资就是让你身边有一些“合适”的人——他们会成为你能找到的最好的资源。这些人可能已经做过你打算做或者想学着做的事情。如果你想做一些事情，最好找到那些做过这些工作的人，或者至少找到一些真正睿智的人，能够告诉你该怎样完成这些工作。尽可能和他们多待些时间，通过相互帮助来向他们学习。与这些高水平的人相处，会让你学到很多东西，不论你的工作是什么，这都会让你表现得更出色。

这是一个很好的想法，但是要想与那些顶尖人物接触可能很困难。像 Martin Fowler、Kent Beck 和 Pragmatic Programmers 的作者等大师级人物，并不是我们大多数人有机会见到的，不过我们可以看到他们的书、文章和演示文稿。所以要开始读书。一个月读一本书应该不会太费劲。但是不要就此止步，接下来可以学习一种新的编程语言，或者研究一个不同的开发过程。在学习和读书时，要想办法把这些新思想应用到当前的工作中。这样一来，你不仅能帮助你的公司提升，更重要的是，还可以让你自己得到提高。

要让自己开放地接受新思想。不要闭塞，应当想办法把这些新思想应用到你现在的工作中。也许你会放弃，并声称某种新思想不适用，这样做当然更轻松，不过我们的目标是学习采用另外一种不同的方式考虑问题。要打破条条框框（或者至少建一个更大的框框）。多学习掌握一些看似不太相关的概念和思想。

通过对环境和过程进行分析和评判，你可以找出弱点。也许这会帮助你对这个项目或者下一个项目做出改进。但同时你也练习了一种新的思维方式，不论在哪里工作这对你都会有好处。大多数人从来不曾了解这个概念，擅长的人更是少之又少。

所以，读完这本书的每个实践后，请停下来，花 5 分钟时间试着想想有没

有办法在你今天做的工作中具体运用各个概念。你可能不假思索地回答：无法做到。这种回答当然最不费劲，不过要记住，不能这么懈怠，你应当更加努力！如果你自己找不出一种方法来应用这个概念，可以找一位同事来帮忙。如果仅凭自己的双眼看不到，可以借助别人的视角来了解。不论在什么领域，知道如何利用同事的经验，绝对是高手特有的标志。

希望你掌握从本书（和整个 Pragmatic Starter Kit 系列）读到的内容，想办法在工作中应用每一个概念。你会看到读这本书给你带来的直接好处，其中最大的好处就是你会学习如何真正加以应用。

希望你喜欢！

致谢

首先要感谢 Andy 和 Dave 让我们为 Pragmatic Bookshelf 写一本书。能为你们写一本书真是荣幸。Andy，你做的已经远远超出了你的份内职责，甚至花好几个小时与我们一同修改手稿，给我们上了一堂写作速成课，尽管有时对我们的写作天份……有些发愁（我们确信你肯定失望过）。Dave，我们曾经问过很多关于图书生成系统和标记语言细节的问题，感谢你花那么多时间回答这些邮件。非常感谢你们二位！

我们有很多非常棒的审校人员，另外还有很多人做出了贡献，他们详细而且有建设性的反馈确实意义重大。没有你们投入的时间，没有你们丰富的经验，这本书不可能出版。Susan Henshaw 和 Jim Weiss 花了大量时间审校我们粗糙的文字，而且读过不止一遍。谢谢你们。

还要感谢 Mike Clark、David Bock、Ken Pugh、Dominique Plante、Justin McCarthy、Al Chou、Bryan Ewbanks、Graham Brooks、Grant Bremer、Guerry Semones、Joe Fair、Mark Donoghue、Roberto Gianassi、Rob Sartin、Shae Erisson、Stefan Schmiedl 和 Andy Lester。你们当中很多人曾忍受过这本书很早的版本，最近我们重新读了你们原先读过的版本，对你们曾经遭受的“折磨”我们深表歉意。说真的，所有反馈都很棒，正是有了这些反馈，这本书才得到了如此显著的改进。

在我们的职业生涯中，曾经与很多人共事过，其中有些人对我们的工作以及这本书产生了直接的影响。我们要特别感谢 Jim Weiss、Randy Humes、Graham Wright、Flint O’Brien、Toby Segaran 和 John Wilbanks。还要感谢我们现在的经

理 Oita Coleman 对我们的鼓励和支持。我们很幸运，能够在 SAS 这样的世界级公司工作。

如果没有敏捷开发社区的智慧，没有大家出色的作品，这本书绝无可能问世。我们读过 XP、Scrum、Crystal 以及很多其他软件领域专家的书和文章。没有你们辛苦而忘我的工作，软件行业可能还在黑暗岁月中挣扎。也许我们还没有完全走出黑暗，不过起码正朝着正确的方向前进。你们孜孜不倦的工作让大家受益匪浅。

开源社区共同为全世界提供了这么多非凡的工具和想法，同样要向你们表示感谢。正是因为全世界开发人员的无私奉献，我们这里讨论的大多数工具才可以免费使用。在此要特别提到 SourceForge 团队和 Apache 软件基金会。你们提供的服务和工具不仅让我们提高了生产效率，还改变了整个世界。

最后，也是最重要的，要感谢我们的主，耶稣基督，愿主永得荣耀！

► Jared Richardson 和 William Gwaltney

我的妻子 Debra 为这本书投入了大量的时间和精力。甚至有好几个星期 Debra 花在这本书上的时间比 Will 和我还要多。其余的时间她既做母亲又做父亲，才让我得以安心地完成这本书。我发自内心地相信，如果没有她的帮助和支持，我绝对不可能完成这个工作。谢谢你！

我的孩子们，Hannah 和 Elisabeth，很多个夜晚和周末爸爸都把自己锁在办公室里一心扑在这本书上，感谢你们忍受了这一切。谢谢你们的理解和爱！

► Jared

非常非常感谢我的家人，很多个漫长的夜晚和周末我都在写书，谢谢你们能理解，还要忍受我在不顺的时候发脾气。是你们让一切付出都变得有意义。

► William

目　　录

如果我们坚持不懈，那么，优秀就不再是一种行为，而成为一个习惯。

▶亚里士多德

第 1 章 绪论

如今，很多软件开发人员都很困惑。他们夜以继日、废寝忘食地工作，可是他们的团队还是无法顺利地完成项目。不是他们的努力不够，也不是愿望不强烈：团队里的每一个人都希望能干净利落地把项目搞定，不过大家都不知道如何共同努力完成工作。你很难有时间坐下来静心读点东西，做些试验，得出哪些做法可行以及如何在你的工作室合理运用。大多数人都太忙于手头的工作，而无暇顾及这种研究。

本书就这样应运而生。这本书汇集了大量基本而实用的建议，这些建议已经在这个领域的多个项目以及大小公司中得到充分证明。我们的亲眼所见和亲身经历可以证明这些方法确实可行。我们不同于那些只在公司出出进进几个星期就离开的顾问，而是天天都在这些公司全力工作。我们并不只是提出一些听起来不错的想法，就匆匆转向下一个预约项目。如果这些想法没有效果，我们还会留在公司查看为什么会失败，哪里出了问题。另一方面，我们必须等到情况好转，工作能够顺利进行下去。

我们介绍的想法有些是从一些众所周知的软件方法中得来的，我们会尽量指明它们的出处。另外一些想法则是我们用热血、汗水和泪水“凝结”而成的。我们尝试过很多工具、技术和最佳实践，如果可行就会将其保留；如果失败，就会断然地放弃。在这里你几乎看不到我们通过盲目摸索得来的一手经验（尽管这是好东西）。相反，我们会“站在巨人的肩上”，精选出这个行业最睿智的思想，把它们转化为你将看到的文字。

如今，50%~70%的软件开发团队并没有使用那些基本的、众所周知的软件

实践（[Cus03]）。很多情况下，这并不是因为他们不知道要做什么，而是因为他们不清楚该如何立即开始运用这些实践。我们会告诉你如何向管理层推销这些想法，给出能够让你迅速上手的明确的实用步骤，然后指出要注意哪些警告信号以免脱离正轨。

本书由“一线”开发人员倾力编写。这本书凝聚了我们在不同公司的实践中得出的经验（从初创的小型公司，到全世界最大的私营软件公司），而不是刻板的理论。这是一个不依赖具体方法论的指南，更注重如何让项目顺利完成。

我们努力使这本书沿袭 Pragmatic Bookshelf 图书一贯秉承的风格：实用、简洁，轻松阅读。希望能将这一风格发扬光大。

1.1 习惯性优秀

为什么这里要引用亚里士多德的名言？“如果我们坚持不懈，那么，优秀就不再是一种行为，而成为一个习惯。”能够生产出一个或一些不错的产品算不上优秀。优秀体现在我们每天所做的点点滴滴，也就是我们的习惯。一流的产品只不过是好习惯的副产品。

把这句名言应用到我们自己身上（包括工作和个人生活中），要求我们能够认识到生活实际上只是习惯的副产品，因此最好审慎地选择我们的习惯。大多数人会毫无计划地草率陷入当前的工作模式而不能自拔，这有很多原因：也许你原先就是这么学的，或者这是你的老板一贯的做法，诸如此类。但是，我们确实能做得更好。

要有意识地搜寻好习惯，并把它们加到你的日常生活当中。

可以做个小实验。找一种要研究的开发方法，挑选一个看上去对你来说不错的习惯（而且这个习惯可以单独运用）。实际使用一个星期。如果你喜欢，而且看来有好处，就继续用上一个月。不断实践这个新习惯，把它变成为你生活模式中很自然的一部分，然后再选择另一个新习惯重新开始这个过程。如同你一砖一瓦地建造地基那样，一次一个新习惯重复这个过程，你就正在为优秀奠定坚实的基础。有时有些做法并不适用于你的环境，请大胆舍弃这些方法，也不要因为某个实践很有名或者很流行就勉强地把它保留下来。要走你自己的路，根据自身情况做出正确的选择。

“我们的每一天怎样度过，一生就会怎样度过”。[1]既然如此，就必须仔细考虑如何度过我们的每一天。

技巧 1

选择习惯

不要偶然地养成某些习惯，要有意识地主动选择习惯。

1.2　务实观点

这本书并不是一个学术著作，单纯地分析为什么有些做法可行或者有些做法不可行，也不会简单地罗列一堆实践和方法供你选择。

相反，这本书会提供现实的软件项目中比较有用的方法。我们会先引入一个新工具或实践，具体加以运用，弄明白它是否可行。接下来，会把那些可行的工具保留在我们的软件开发工具箱中，并“随身携带”。最终看来，我们所做的确实很奏效！希望这些工具和实践也适用于你。

我们参与过很多小型初创公司的工作，这些小公司条件有限，并不是某种方法“正确”就能够加以使用——他们无法做到如此“奢侈”。我们所处的环境要求我们找出能够立即投入使用的解决问题之道。我们也在较大规模的公司工作过，这些公司拥有丰富的资源和技术。但我们发现，大公司也不会因为某个工具很优秀或者某个专家很推崇就使用这个工具。他们想要的是能够快速而经济地解决当务之急的方案。所以我们会有取舍地挑选合适的习惯，最终建立一个足够通用可以移植，并且仍能有效地解决具体问题的工具箱。这本书汇集了我们使用过的很多好习惯，它们也会对你的工作室产生深远影响——结果可能是惊人的。

为了便于说明，下面来讲一个有关两个软件工作室的“双室记”故事（因为篡改了《双城记》的名字，在此向查尔斯•狄更斯致歉）。

第一个工作室简直一团糟。他们购买了非常昂贵的源代码管理软件，但是从来没有安装过。结果，他们甚至把原本打算向潜在客户展示的演示版的源代码都弄丢了。没有人能确定产品中应当包含哪些特性，但不管怎样，整个开发团队还在兢兢业业地工作着。开发出的代码很不稳定，每 5 分钟左右就会崩溃

① Annie Dillard，美国作家，诗人，1975 年因其散文《溪畔天问》获得普利策奖。

一次（经常还会在关键时刻出问题，比如说在现场演示过程中）。这种混乱严重影响了大家的士气：公司例会的气氛经常急转直下，演变成一场场争吵。有些开发人员成天躲在自己的办公室里，以躲避这种场面。总之，这实在是一个糟糕的工作环境。所有人都知道存在严重的问题，但是没有人能解决。

第二个工作室情况就好多了。开发人员的人数与第一个工作室相当，他们在分头同时开发三项主要产品。这些项目的代码都放在一个源代码管理系统中，只要代码有变更，就会自动重新构建并测试。整个团队每天召开例会，尽管简短，但非常专业而有效。每个项目都有一个主计划，因此每个开发人员都知道要完成哪些功能。他们遵循采石工人的信条：*尽管我们只是采石头，但脑海中必须想象着最终建造出的宏伟教堂*[HT00]。也就是说，每个人都能在一个更大的框架范围内充分施展他们的专业技能和才华。他们会按时交付问题最少的产品，这些产品都经过精心设计，因此很稳定。

最让人惊讶的是，这两家公司实际上是同一个工作室，这两种状态相隔不超过 6 个月，区别仅是之前没有应用这本书中的原则，而后来将这些原则充分加以运用。（你应该已经猜到了，对不对？）情况好转之后，CEO 说我们引入了一种“卓越的气氛”，他“甚至没有认出这个地方”。这是我们最近参与工作的一家公司，就像书中为你展示的一样，我们为他们应用了这本书中的原则。这种巨变也是促使我们写这本书的一个原因。

我们发现了这些方法，并将其应用于各种规模的公司，从只有 4 个人的小型初创公司到全世界最大的私营软件公司 SAS。坦率地说，这些原则居然能适用于各种规模的公司，让我们也感到惊讶。

可以认为这些理念是得出优秀产品的基础。如果你愿意在前期投入时间建立这么一个很好的基础，在余下的产品生命周期中就会大有收获。当然，如果充分采用这些实践，启动一个项目就会容易得多。就像一幢房子毁坏的地基，有些方面可能很容易修补，而有些则是更深层次结构上的问题，需要下很大功夫才能修复。

目前你的项目可能正在进行当中，但什么时候开始采用好习惯都不晚。你可以把这些想法引入到一个现有的项目中，很快就能有所收获，我们会在最后一章全面介绍有关方法。

1.3 路线图

我们把想法整理为三部分：基础设施、技术和过程（见图 1-1）。这些方面会直接影响你的团队能否始终如一地交付客户想要的产品。

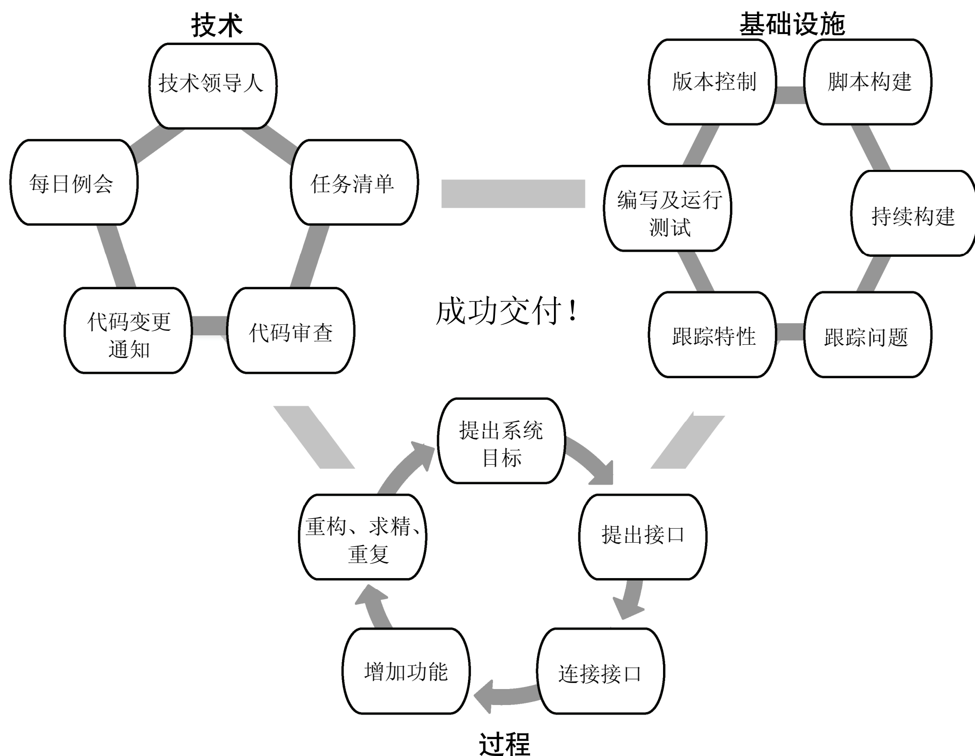


图 1-1 如何构建优秀的产品

1.3.1 基础设施

在第 2 章中，我们会介绍一些软件工具，它们能使你和团队更加轻松地完成工作。例如，一个好的源代码管理系统可以保证项目的“重中之重”（即源代码）安全可靠。自动构建系统可以根据需要随时随地提供可重复的构建。我们还会讨论如何跟踪记录 bug 报告和特性要求，以及向全世界发布你开发的产品的那一刻还可能会出现什么其他问题。最后，我们还会介绍一个很好的自动化测试框架，可以让你放心代码确实在老老实实地工作。

1.3.2 技术

在第 3 章中，我们将介绍你和你的团队每天可以使用的一些具体实践，做到“更巧地工作，而不是更玩命地工作”。我们会介绍如何在团队中引入一位“技术领导人”，使你与外部世界隔离，只得到你需要知道的信息。你可以使用“任务清单”来组织自己以及整个团队的工作，保证团队不脱离正轨。你的团队是不是沟通不畅？是不是不知道谁在做什么？如果是这样，就应当开始召开“每日例会”，保证所有人都达成共识，同时借机采纳团队成员的想法。简短的“代码审查”有助于利用同事的专长，还可以与别人分享你的才能。一旦审查结束，就要利用“代码变更通知”向团队的其他人员展示你做了哪些工作。

1.3.3 过程

如果一本关于软件开发的书没有展示作者最钟爱的开发方法，就谈不上完整，这本书也不例外。所以我们特别增加了一章，取名为“曳光弹开发”（Tracer Bullet Development）。使用曳光弹开发方法时，可以创建一个端到端的工作系统，其中大部分组件都是中空的对象，然后填入缺少的部分，使它成为一个真正实用的系统。这对于分解大型项目并让团队并行完成各部分工作很有好处，而且利于自动测试。

1.3.4 常见问题及解决办法

最后我们会指出一些可能出现的常见问题（和危险信号），并给出实战建议，告诉你如何利用我们在书中介绍的工具、技术和过程来解决这些问题。这里的很多问题我们自己就曾遇到过。有些当时解决了，有些则是事后才知道该怎么办（毕竟，人们经常是事后诸葛亮……）。希望我们的经历能避免你重蹈覆辙，不会犯我们犯过的错误。

1.3.5 哪些内容没有谈到

人员配备和需求收集不在我们的讨论范围内。好的人才是项目中最重要的部分，他们的作用胜过工具、技术和过程。不过，组织和维持一个优秀的团队是一个值得单独用一本书（甚至一系列书！）去探讨的话题。我们不打算讨论这个方面，而会重点讨论如何充分利用并且进一步发展团队已有的技能。

类似地，了解产品需求也是一个需要深入研究的主题。收集需求有很多方

法，从简单的便条卡片，到需要许多检查和平衡的复杂系统。这个主题仅用一章根本无法说明白，我们并不打算去解决这个难题，而是选择了另一种做法，给出一些足够灵活的想法来处理不断变化的需求，而不论需求是从哪里得来的。本书中的想法适用于各种项目，包括需求从不改变的项目，也包括需求不断变化的项目。所以不论你是利用一堆 3×5 的小卡片了解需求，还是根据一个 10 000 页的庞大合同分析需求，都可以使用这些想法。

我们尽量保证讨论的适用面足够广，以便在任何工作室利用任何技术都可以采用这些想法。正是由于这个原因，我们没有特设章节来讨论安装技术或代码优化工具。

1.4 继续前进

对于这里给出的理念和习惯，希望你能本着它们的内在精神（也就是务实地）加以使用。先阅读，然后动手尝试。根据你的环境，保留那些适用的想法，其他的断然舍弃。

读完每一节后要停下来，想想你是否正在采用刚刚读到的方法。如果没有，那么请读一读“如何起步？”；如果确实采用了这种想法，那么可以读一读“我做得对吗？”或者“警告信号”来确认一切正常。

1.5 怎样读这本书

如何学习这本书取决于你在项目中承担的角色。很自然，如果是一个开发人员或测试人员，那么你学习这本书的角度肯定与你的团队领导人有所不同，但不论你是什么角色，都能从这本书中收获很多。

1.5.1 开发人员或测试人员

如果你是一线从业人员（或程序实现人员），那么可以从头到尾读完这本书。每一节都包含一些实用的想法，不论是普通员工还是团队领导人，都可以在每天的工作中使用这些想法。有些开发人员由于不是团队领导，往往会跳过强调团队的章节，这其实很不好。大多数团队环境都是由团队成员的诉求或者他们的直接经历造就的。要把你自己放在一个更高的位置上，知道哪些工具、技术和过程会为你的工作室带来正面影响，并且能为你的每个提议给出充分的理由。我们经常听到开发人员因为某个工具或技术展开争辩，因为“那才是正

确的做法”。管理层绝对不会被这种争辩所左右，实际上，争来争去可能还会适得其反。提出一个想法之前，一定要明确了解它对团队有什么好处。

看看这样两个提议，一个是：“我们需要 Acme Code Systems 的一个源代码管理系统，因为这个系统很不错，所有人都在用。这是一个最佳实践！”另一个提议是：“我们应该有一个源代码管理系统，有了它我们可以访问以前的版本、撤销特定的代码变更，还允许开发人员安全地处理并行代码树。这是保障公司开发投资最容易的办法。Acme Code Systems 提供了一个不错的产品，我们应该了解一下。Joe 和我已经用了几个月，确实对我们的工作效率有很大改善。它会在以下这些方面对我们提供帮助……”在这两个提议中，哪一个更能让你接受呢？

1.5.2 项目团队领导人

可以用这本书对团队的环境和工作流做一个审查。（你已经在不时地做这个工作了，对不对？）请利用这个机会重新检查你的团队是如何工作的。有没有一组基本工具可以满足你的基本需求？团队的技术能不能构建可靠的产品，培养可靠的开发人员？有没有一个明确定义的简洁的过程？

审查你的团队如何工作时，一定要考虑每一项的关联性。有没有用到一些原先合适但现在已经失效的工具或实践？

你听说过这样一个故事吗？一位女士烧火腿时总是先切掉三分之一。有人问她为什么这么做，她说她妈妈总是这么烧火腿。而她妈妈也说她的妈妈一直是这么做的。最后问到她的外祖母时，老人家说她年轻时家里的锅不够大，没办法放下整只火腿，所以她总是先切掉后面一段，时间长了就养成了习惯。

一定要保证你的习惯是根据当前的需求形成的，而不是因为去年的项目就是这么做或者只是因为“老奶奶式”的怪癖。

确保你的团队可以得到当前需要的工具、技术和过程。要了解哪些可行以及为什么可行，这是有效地指导团队的唯一途径。我们会在每一节介绍一些技巧，帮助你着手实践，还会给出一些警告信号，提示存在问题以防完全失控。

1.5.3 经理（或当事客户）

更高管理层只需询问正确的信息，就能对团队如何工作产生很大影响。这本书会告诉你团队应当使用哪些重要工具，以及你应询问哪些类型的问题。例

如，如果你要求得到上一个版本的修正清单，实际上就是在说你希望这个信息得到跟踪。读每一小节时，找出要求团队领导人提交的可交付产品（deliverables），这会引导他们的工作不脱离你希望的方向。不过，一定要审慎地提出要求，你肯定不希望你的要求导致团队毫无意义地瞎忙，而希望通过精心设计的要求做出正确的引导。

因为你不必完成开发人员的日常工作，所以可能会跳过“如何起步？”小节，不过你肯定想了解每个主题是什么以及为什么那样做。

1.5.4 个人组成团队

这本书中几乎每一个理念都经过团队成员、整个团队以及经理的实际运用。团队成员通常是第一个使用这些实践、证实其价值，然后与团队分享的人。我们自己反复做过这个工作，也见过其他人做过，你也同样可以。有人就这样做过，下面听听他们的故事。

CafePress.com 敏捷支持系统的快速发展

Dominique Plante 和 Justin McCarthy

去年年初我们开始在 CafePress.com 工作，那时管理层对于采用敏捷实践可谓热情万丈，不过要想信心十足地做出改变，开发环境还缺乏必需的基本支持系统。

“Enter the Create and Buy”项目——这是 CafePress 核心产品的一个扩展，允许客户轻松地设计和购买定制的商品（如 T 恤、杯子等）。这个项目首次尝试除了 Web 表示层外还引入一个明确的业务和持久存储层。业务和持久存储层大部分都设计为先行测试，使用 NUnit 框架来编写开发测试。同时，我们引入了 NAnt，为 Web 层中使用的类完成可重复的编译和部署。接下来，我们利用 CruiseControl.NET 基于 Subversion 代码存储库的每次签入完成持续集成（即编译和运行测试）。最后，委托 Chicken Little（一个规模很小但高度可见且可听的工作站）运行 CCTray 来提供构建状态通知。

Subversion、CruiseControl.NET、NAnt 和 NUnit 之间的互操作性帮助我们建立了一个友好的协作环境，而不再频频出现有争议的供应商分析或购买决策。另外，这些支持系统都是由开发人员主动提出的，而不是在管理层要求之下被迫接受的。

由于我们完成了这些自动化，团队开始壮大，很多新的团队成员促进了我们的测试套件和项目自动化工具的成熟过程。最新的升级包括 100%脚本开发环境的创建以及自动化测试环境部署，不过回头来看，最常使用的目标仍然是 nant test。

在这些工具出现之前，我们每天的沟通主要是向大家广播构建失败问题或 API 变更通知。现在 CruiseControl.NET 会处理我们的“伙伴构建”，开发人员了解并履行承诺，会保证构建不被破坏。任何人都不愿意由于一个错误的代码签入而导致工作中断。有了这

些支持系统，我们的交流很自然地转向软件设计和实现，而不再为如何消除环境方面的问题而困扰。

我们前期的所有努力没有白费，在此帮助下，代码经过测试并即时交付，而且每次 Chicken Little 发出“叫声”通知变更时，我们原来的投入就会产生回报！

我们认为这才是改变的最佳方式。我们从来不拒绝经理引入改变，但是我们发现最好而且最可行的改变还是来自于一线人员。具体做事的人应该更了解需要解决哪些具体问题。

所以我们认为，不论你是经理、开发人员、测试人员还是技术领导人，都请“使用这本书”。找出你的工作室（或你个人的工作）中缺少的环节，看看这本书是如何让你现在的生活更轻松的。



小乔爱问……

什么是敏捷?

敏捷（agility）是指软件团队能够很快适应不断变化的条件的能力。这有时意味着要重新设计以适应变化的需求，有时则意味着快速应对新的 bug 或迅速采用新的技术。总之，敏捷团队更关注结果而不是形式。关于敏捷软件的更多信息可以参考 http://www.agilemanifesto.org/。

下面这段话是从这个网站摘录的，对敏捷观点做了很好的总结。

“我们正在提出更好的软件开发方法，我们自己在使用，同时还帮助其他人使用。通过这些工作，我们意识到：

- 个体及互动比过程与工具更有价值
- 可用的软件比冗长的文档更有价值
- 客户协作比合同谈判更有价值
- 对变化的响应比遵循计划更有价值

也就是说，我们认可上述右边事项的价值，但我们更加重视左边的事项。”

……增加一个功能特性的成本并不单单是为这些功能编码所花费的时间，这个成本还应该包括对将来扩展所设置的障碍。……这里的诀窍是要挑选不会彼此冲突的特性。

►约翰·卡马克

第2章 工具和基础设施

有两个人（Mike和Joe）都想盖一幢房子。一个人（Mike）先花了很长时间来选购工具并学习怎么用这些工具，当然也花了不少钱。另一个人（Joe）就用他现有的工具（一个榔头和四个螺丝刀），立即投入工作。并不奇怪，Joe的房子起步更快。就在Mike学习如何使用空气压缩机和钉枪的时候，Joe已经在用榔头钉钉子了。不过，当Mike完成了他的学习过程，开始着手盖房子时，很快就把Joe落在了后面。Mike投入时间来学习如何使用工具，因此可以在更短的时间内造出更棒的房子。另外，如果要盖下一幢房子，谁会更快呢？对于这个问题的答案，应该没有人会存在疑问吧。

与Mike一样，有很多工具可以任由我们挑选。我们可以像Joe那样，满心只想着"立即开始工作"，拿起自己熟悉的工具就贸然行动；或者也可以先退后一步，仔细考虑如何开展工作。也许这一章谈到的一些工具你的工作室里还没有，或者虽然有但没有得到很好的使用，给你留下了糟糕的印象。建议你仔细研究这一章，看看我们讨论的工具对你的日常工作有没有正面影响。

下面来看Fred的一天。Fred是一个很有代表性的开发人员，在一个很普通的工作室工作。他的工作经历很大程度上受所在环境的影响——也就是工作室的工具和基础设施。

没人知道Fred遇到的麻烦

Fred早上一上班，就看到Wilma发来的一封邮件，告诉他头天晚上她对代

码做了一些修改。由于 Fred 需要这些修改，所以他把 Wilma 的代码从一个网盘复制到自己的机器上。经过一个小时的修修补补，他终于能够在自己的机器上成功地编译 Wilma 的代码了。他又花了几分钟再次检查 Wilma 做的修改，看起来没什么问题。然后 Fred 查看昨天的记录来回忆目前的工作。他正在编写一个新特性，马上就要完成了。他已经为此奋斗了三天，觉得应该能在午饭前大功告成。

Fred 一心想尽快完成工作，他打开代码编辑器，发现他之前所做的修改居然不见了！就像一盆冷水浇得他透心凉。Fred 突然意识到他复制 Wilma 的代码时覆盖了自己的工作。三天的辛苦工作在短短 30 秒里就荡然无存，而且根本没办法挽救。“唉，”Fred 心想，“这不是第一次了，也不会是最后一次……这就是这项工作的危险之处。”

这时，销售部的 Richard 来查看一个 bug 修正的进展情况，他们上周就提出要求修正这个 bug。

Fred 很不好意思……因为急于增加这个新特性，他居然忘了处理这个 bug。Richard 对他的拖延很不满意，不过 Fred 答应当天下班前一定完成这个工作。

Fred 终于在下午很晚的时候修正了这个早该处理的 bug。他刚准备把更新的代码发送给客户，突然意识到这个客户用的是产品的前一个版本，而不是他刚修正的这个版本。办公室里的其他人都准备下班了，Fred 却不得不打电话给妻子取消原定的晚餐计划，开始在网盘上搜索老版本的代码，以便把刚做的修正移植到产品的前一个版本中。

经过大约一个小时的努力，他终于找到了那个版本的代码；然后又用了一个小时才让这个代码在他的机器上运行起来；接下来又用了一个小时将代码修正移植到这个老版本的代码上。Fred 总算搞定了。

这真是漫长的一天，不过他为自己这么长时间的努力工作感到自豪。在 Fred 看来，只要能让公司成功，再辛苦也愿意！不过他的家人并不总是表示理解。

第二天，客户发现 Fred 在产品中意外地引入了原先的一个 bug，另外还引入了两个新 bug。

你的一天有什么不同

下面假设你处在 Fred 的位置上。早上一上班你发现 Betty 发来了一封邮件，

告诉你她对代码做了修改。你的工作室使用了源代码管理（Source Code Management，SCM）包，将所有代码变更都存储在一个中心存储库中，然后根据需要下载变更代码。所以你只需要向 SCM 请求 Betty 更新的代码，并合并到你的机器上现有的代码中。如果存在冲突或者代码覆盖，SCM 会做出警告而不至于丢失代码变更。

接下来，你只需执行一个单行构建命令，Betty 做的变更就能与你的代码一同顺利通过编译。由于你的工作室使用了一个标准构建系统，所以 Betty 会与你采用相同的方式构建代码。与 Fred 不同，你根本不用为了在你的机器上完成代码编译而多花一个小时。

你刚要开始工作，销售部的 Richard 来询问上周要求的 bug 修正情况。与 Fred 一样，你起初也把这件事忘得干干净净，不过你有一个习惯，会定期查看团队使用的 bug 跟踪软件的每周报告，它在星期一已经提醒过你。所以你已经修正了这个 bug，而且为客户准备好了更新版本。Richard 非常满意，很高兴地离开了，而且对你有了一个很好的印象。

再回头说说修正 bug 时的情形。你必须找到相应的源代码，因为这个客户使用的不是当前版本。你请求 SCM 软件提供较早的那个版本，SCM 软件满足了你的要求。根本不用花时间在网盘上搜索代码，当然也不会因为较早的代码树意外地重新引入 bug。

即使使用了一个有 bug 的代码树，构建过程还包括运行一个自动化测试套件。这个套件会运行一组轻量级的测试，专门用来捕获常见的 bug。如果重新引入了 bug，在客户发现之前你就能将其捕获。

这一天结束时，你按时下班，与家人共进晚餐。Fred 则不然，他工作到很晚，他的妻子又生气了。你已经完成了这三天的工作，成功地增加了新特性，明天上班时就可以转向下一项任务了。你比 Fred 更有“工作成效”，不过这并不是因为你更聪明或者更投入——你们都是很努力很聪明的开发人员。

你和 Fred 之间只有一个区别：你的工作室使用了一些关键的工具，而 Fred 的工作室没有。有了这些工具，就为开发提供了坚实的基础设施，可以轻而易举地解决很多让人头疼的常见问题。

避免Fred的陷阱

Fred 太忙于修正 bug，四处去“救火”，以至于根本没有意识到还有更好的解决办法。每个成功的项目都有一个强大的基本框架做后盾，也就是基础设施。这包括团队构建产品使用的工具和实践方法，如源代码管理系统和构建工具。具体使用的工具可能有所不同，但是对于所有项目来说，这些工具的种类都相差无几（见图 2-1）。这一章将会介绍每个项目都应使用的一些基本工具。

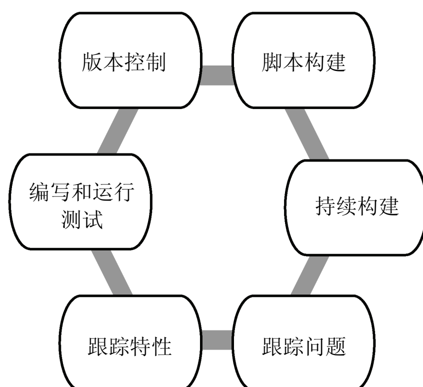


图 2-1 工具和基础设施

这一章中有些观点看上去可能太过浅显，尽管如此我们还是包含了这些内容。一个内容很浅显的观点，并不意味着它已经得到广泛使用，也不表示使用的方法就正确。

有些团队可能在大多数方面都“配备精良”，但在另外一些方面却存在严重的盲点。尽管他们在某些方面大把大把地花钱（因为供应商总是声称这些“超级工具”无所不能），不过可能遗漏了几类重要的工具。

要记住，每读完一个实践都要停下来，把你刚刚读到的内容与你目前的工作做个比较。如果没有用到我们谈到的工具或想法，就问问自己，为什么没有用？是因为你从来没听说过，还是因为你以前用得很少、没什么经验，或者只是因为这个工具对你不适合？如果确实在使用这个工具或想法，那么你的方式方法正确吗？如果不正确，怎样才能加以改善？

1 在沙箱中开发

你与其他团队成员如何共享代码？相当多的团队从来都不明确回答这个问题，而是拿出一个巨大而老旧的共享磁盘，里面存放着他们的所有源代码和其他一些文件。任何开发人员所做的任何活动——从简单地编辑一个文件到编译代码——都会立即影响团队中的每一个开发人员。他们不断遭遇令人不快的意外。

这就像在感恩节拥挤的厨房里，每个人都在往一锅大杂烩中扔东西，弄得一塌糊涂，这实在是一个让人沮丧的工作环境。尽管很多团队还顽固地采用这种方式运作，但你完全可以采取一种更安全、更专业的立场。这会对你的工具和基础设施产生深远的影响，所以从一开始就应该有正确的方向。

只需记住一个基本原则：在你准备好之前，要与其他人隔离，使他们不会受到你的工作的影响。基于此，我们把这种方法称为沙箱开发（sandbox development）：每个开发人员都有自己的沙箱，可以在沙箱中尽情尝试，而不会干扰其他开发人员。

这听起来可能很容易，特别是当把这个原则表述为隔离源代码（参见实践 2）时。不过这里真正的难点在于，要记住这个原则适用于所有资源，不光包括源代码，还包括数据库实例，以及你依赖的 Web 服务等。

你自己的开发机器应当特别针对你，可以提高你的生产效率[1]，而不应对全局构建过程有任何贡献——别人做任何事情不必直接依赖你的机器。

不过其他开发人员如何得到你的代码呢？代码可以通过存储库（repository）共享。可以把存储库想成是一个巨大的共享磁盘，但会由一个“图书管理员”管理。这个管理员要确保每个人可以得到他们需要的任何文件（或者其他资源）的正确版本，另外所有人都能正常工作而不会相互干扰。每个开发人员使用一个软件工具来完成文件的签入和签出（就像在一家真正的图书馆里），从而可以在本地处理文件。

在你自己的开发机器上，可以编辑源代码文件的本地副本，完成文件的编译、构建和测试，所有这些都与其他团队成员完全隔离。如果你在开发期间需要使用数据库、Web 服务器或者其他资源，一定要确保只有你一个人在

① 这意味着，不同的开发人员完全可以使用不同的代码编辑器甚至不同的集成开发环境（IDE）。

使用这个资源。如果你对完成的一段代码感觉满意，要把它再签入放回到存储库中。

不过，客户如何得到最后完成的产品呢？除了开发机器和存储库外，还有一个构建机（build machine）。构建机是一个无人照管的服务器，它只是从存储库得到所有最新的源代码，反复地构建和测试。构建的结果就是产品发布。

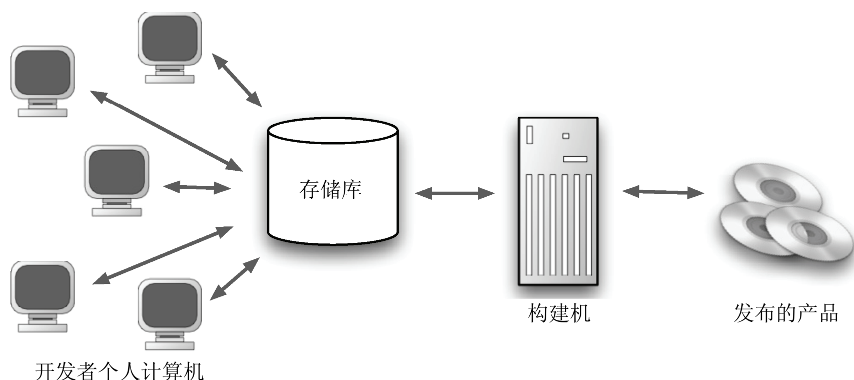


图 2-2　沙箱开发配置

大多数情况下，每次构建之后的这个发布版本都会被丢掉，不过有时这也可能成为最后交付给客户和最终用户的产品。不论是每天上午 10 点的例行构建还是经过数月艰苦奋战才得到的最后发布版本，都会采用同样的方式构建。

构建方式之所以会始终如一，是因为构建机是一个独立的实体：它不会出于任何原因查看个人的开发机器。构建的输入是存储库，整个构建过程的输出都来自一个指定的构建机。只要开发人员循规蹈矩，这个系统就能很好地工作。

技巧 2

留在沙箱里


小乔爱问……

发布版本来自哪里？

你的构建机可能是也可能不是构建发布版本的机器（发布版本就是交付给客户的代码）。不过，构建机和交付产品构建机都使用相同的脚本，使用同样的存储库作为输入源，很多方面都完全一样。

它们也存在一些区别，交付的构建版本会在存储库中创建一个新的分支或标签，来标志一个已知的发布代码包，或者交付的构建版本有可能把代码包装在面向不同平台的安装程序中。

有时很难做到“留在沙箱里”，特别是当数据库许可证或 Web 服务器端口供应不足时。如果是这样，可以使用一个数据库，但是应为每个开发人员创建单独的实例。或者，如果要求一个数据库只能使用一个实例，那么可以划分数据空间（例如，为 Joe 分配账户 1000~1999 的测试账户数据，而为 Sue 分配账户 2000~2999，依此类推）。尽管这样仍有相互干扰的风险，但毕竟好多了。

对于其他资源，如 Web 服务，每个开发人员在各自的实例上都应当有一个明确的目标（不论是提供服务，还是进行测试）。

有了隔离的基本想法，下面来看要达到沙箱效果需要哪些工具和基础设施。

2 管理资产

大多数公司都有庞大的系统和大量人员专门致力于资产管理（asset management）的工作，也就是跟踪公司的所有有价值的有形资产——计算机、汽车、大楼，甚至小到订书机这样的办公用品。对于一个软件项目，这个任务要稍微简单一些：需要跟踪的只是文件。不过你需要跟踪构建中使用的每一个文件的每一个版本，从项目开始直到结束。

要完成这个可怕的任务，需要一个源代码管理（Source Code Management，SCM）系统，这也称为版本控制（Version Control，VC）系统。这些系统就相当于很能干的图书管理员，它们会跟踪存储库（或数据库）中的所有资产（文件），并协调对这些文件的安全访问。当然这些系统要存储源代码，不仅如此，还要将所有其他支持文件归档，如图像、构建脚本、XML 片段、文档以及每个人都离不开的很小但很重要的 Perl 脚本。

有了一个合适的 SCM 系统，你就可以做到以下几点。

- “为团队在项目范围内提供一个‘撤销按钮’：没有什么是最终的、无法改变的，错误可以很容易地回滚。[TH03]”
- 可以在多个人同时编辑（或想要编辑）某个给定文件时处理冲突。
- 跟踪软件的多个版本：在有人修正前一个版本中的 bug 时，你可以同时为下一个版本增加新特性。
- 记录哪些文件有变更（什么时间以及由谁变更）。
- 查看历史：可以获得任意一天的工作快照。

如果你的整个开发工作室都被火烧为灰烬，利用一个存储库备份就可以轻松恢复。你应当具备构建整个产品所需的全部内容。如果做不到，那就说明你可能没有正确地使用这个工具。

技巧 3

如果需要就将其签入

有些人可能把第三方产品排除在外，如 Java 运行时库（Sun 公司免费提供）或其他特定产品。他们不在 SCM 系统中存储这些产品。假设你需要用到一个第三方产品，而且这个产品的任何版本都能使你的产品顺利运行，那样倒还没什

么问题。然而，如果你要依赖这个第三方产品的某个特定版本，就要当心了，你得看看另外那家公司会不会继续为你提供和支持这个产品版本。

一般来讲，要“保证构建产品所需的所有内容都在 SCM 中”，对此只有一个例外：那些可以生成的文件不必放在 SCM 中。如果你有 150 个库（存储为 JAR 或 DLL 文件），它们都是由公司内部的产品构建的，把这些库都存储在 SCM 中就没有必要了。你完全可以在需要时重新构建它们，因为 SCM 中有原始代码。不过，如果这 150 个库都来自其他公司，而且没有它们你就无法构建产品，则应当把它们包括在 SCM 中。

另外要考虑的一点是，产品（以及相应的整个公司）往往会彻底消失。如果你的供应商破产或者开源工具的提供者决定不再支持这个产品，你还能生存下去吗？如果一个重要的供应商破产或者一个开源项目终止，你还能正常交付产品吗？产品（包括开源产品和商业产品）经常会出人意料地消失。

你的源代码管理系统一定要包含构建、部署和运行产品所需的一切。如果做不到这一点，实际上就是为了节省磁盘空间而拿开发项目的长期发展在冒险。

这听起来确实有点蠢，不是吗？

有了 SCM，你可以随意使用产品的某个特定版本（当然可以是当前版本），可以回滚特定的变更。如果采用这种方式，你做了很长时间的工作就不会意外丢失。你的工作非常重要，不应被覆盖更不能被完全擦除，要把它存储在一个源代码系统中，从而可以回滚变更并返回到一个已知正确的代码版本。

狗把我的源代码吃掉了

你知道产品的源代码放在哪里吗？我是说全部，每一个版本，你确定吗？你确定团队的其他人也知道吗？

看看下面这个例子。我们加入的一个小网络公司花巨资（多达 6 位数）购买了一套先进的源代码控制系统。（哈，着实体会了一回风险投资的快乐！）不过他们到现在还没有安装这个系统。三个开发人员（没错，只有三个人）分别有不同版本的产品代码，每个版本中都有各自的一堆 bug。

这几个版本都达不到公司为预期客户展示的标准。我们费尽心思，试图协调这三个不同的版本，不过最后还是决定放弃这种努力，选择其中一个版本作

为起点重新开始。

我们安装了 SCM，将这个版本的代码放在 SCM 中，而且在此之后一直使用 SCM。每个人都知道哪个代码是“真正”的产品代码，这不仅让开发人员感到心里踏实，而且产品的质量也得到显著提高。

如何起步

如果你的项目没有使用任何源代码控制措施，那么现在就要把它作为最首要的事情来办。

根据最新调查，美国 IT 企业中约 40%根本没有使用任何源代码或版本控制产品，所以我们知道你也在此列 [Zei01]。

(1) 对几个 SCM 系统进行评估（见附录 B）。我们衷心推荐 CVS 或 Subversion（图 2-3）。

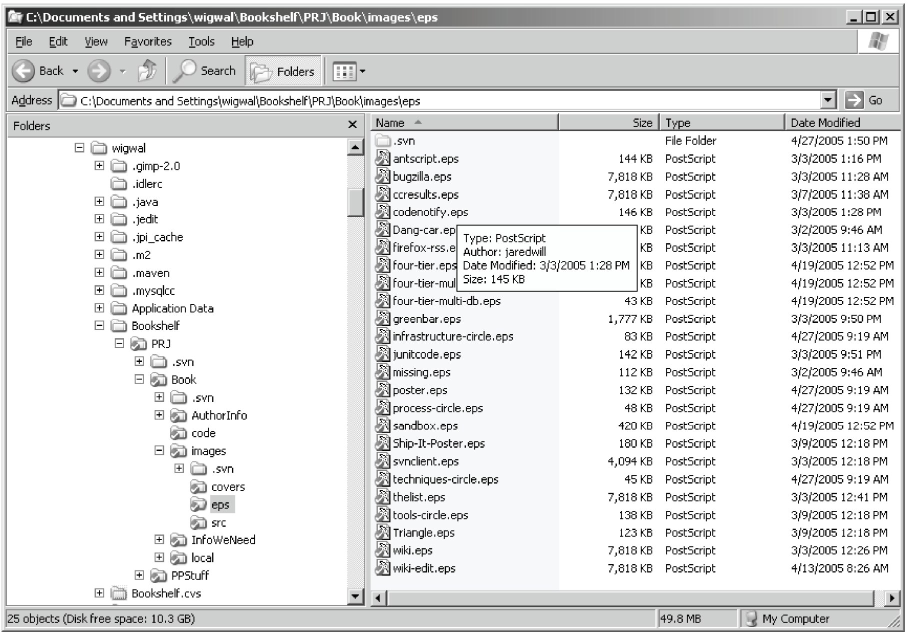


图 2-3　SUBVERSION GUI 客户端中显示的版本变更

这两个工具都是免费的，全世界一些很大的软件公司都在使用它们。如果你做的是商业解决方案，也仍然可以把这些产品作为基准来比较各工具的特性。

(2) 学习如何使用你选择的 SCM。

(3) 生成一个简短（只有一页）的快速入门指南，列出如何使用这个系统完成常见的操作。(警告信号请查看下文的列表。)

(4) 向你的团队展示系统。一定要让每个人都熟悉这个系统。

(5) 导入你的代码和支持文件。

(6) 从此开始把所有文件都放在 SCM 中。

我做得对吗

首先，你有没有积极地使用这个系统？如果隔几周——或者几天——才做一次代码签入，就说明你没有积极地使用系统。这就达不到你本来的目的（备份之前的工作，维护一个准确的、细粒度的历史）。即使你的代码还没有准备好、尚不能提交到生产树，也可以放在这个系统的一个私用或个人区中。

如果工作站硬盘突然崩溃，你会损失多少天的工作？如果达到一两天，那就应该考虑改变工作方式了。

另外，你要花多久才能安装好一个新机器，让它可以投入开发？所有构建脚本和开发资源都签入了吗？

你能快速操作 SCM 吗？如果要花 20 分钟才能签出代码，还要再花 15 分钟才能将修改后的代码放回，你就不会经常做签入。基本的交互操作一定要快。

像获取两个版本的差别之类的操作可能会给 CPU 和磁盘带来沉重的负担。很多公司把他们的 SCM 软件放在一个老旧的“古董”计算机上，这让所有交互都很费劲。

硬件并不贵，真正宝贵的是开发人员的时间。不论需要什么，培训、硬件或者新的 SCM 软件，都一定要保证 SCM 足够快，能跟上你的速度。

你备份了 SCM 的存储库吗？如果今天晚上构建失败，有没有一个非现场（off-site）备份可以在另一个机器上恢复？如果丢失了存储的数据，再好的 SCM 也没有用。一定要确保至少有一个完整的每周非现场备份。

最后，你了解你的系统吗？能不能轻松地完成这些基本操作？下面列出了你起码需要掌握的操作。

- ❑ 签出整个项目。
- ❑ 查看你编辑的代码与 SCM 中最新代码之间的差别。

- 查看特定文件的历史——谁修改了这个文件，什么时间做的修改？
- 根据其他开发人员做出的修改来更新你的本地副本。
- 将你的修改推送（或提交）到 SCM。
- 删除（或撤销）以前推送到 SCM 的变更。
- 获取上周二代码树的一个副本。

警告信号

- 存储库里没有任何活动。如果没有使用，这个存储库就毫无用处。照理说，由于各个开发人员定期地签入和签出，你一天中每个小时都应该能在存储库中看到活动。
- 存储库不完整。如果存储库只包含构建产品所需的部分文件，就难以发挥作用。要特别留意未放入存储库的“试验”版本或测试树，及关键 XML 文件或数据库的内容。
- 访问缓慢。如果要花很长时间来签入或签出文件，人们最后肯定不会再用这个系统了（即使目前在用）。
- 文件丢失或损坏。很多人都知道有些版本控制系统会定期“吃掉”文件（这些系统由一些大型公司开发，这里就不点名了）。最好升级到一个真正实用的系统，比如可以免费使用的 CVS[1] 或 Subversion[2]。

① 见 http://www.cvshome.org。

② 见 http://subversion.tigris.org。

3 建立构建脚本

构建（build）会把源代码转换为一个可运行的程序。取决于计算机语言和环境，这可能意味着编译源代码，或者根据需要打包图像和其他资源。编译和资源打包时使用的脚本、程序和技术结合在一起就构成了构建系统。

要记住，我们现在并不讨论如何在 IDE 中完成编译或构建。大多数情况下，即使只有你一个人使用这个项目，你也不希望用自己的开发 IDE 来编译（如果你觉得听起来有些奇怪，可以参见实践 1）。我们要讨论的是你自己机器上的构建要与构建机上的“官方”构建同步。下面会讲一个小故事，来解释为什么要这样。

Billy 准备构建产品，所以启动了他的 IDE，并打开他认为正确的项目。他让 IDE 重新构建这个项目。IDE 完成构建后，Billy 退出 IDE，并把这个程序复制到他的安装工具目录。然后打开安装程序，指向先前构建的程序，要求构建一个安装工具。

构建了安装工具之后，Billy 运行这个安装工具确保它能正常工作。可惜它马上就崩溃了。Billy 想起来他忘记了一点，没有复制这个产品依赖的第三方部件（Widget）的最新版本。所以他把这些部件的最新版本复制到安装工具目录。第二次构建程序之后，他意识到又忘了一步，没有复制程序使用的图像的最新版本……

感觉就像在跑马拉松，是不是？Billy 辛苦了一晚上，不断地发现问题，只好继续加班，没有任何报酬，还错过了看电视剧《吸血鬼猎人巴菲》的重播。

Bob 今天也在构建产品。他先进入包含代码的文件夹，键入一个单行命令（例如 `ant build_installer` 或 `make all`）运行他的构建脚本。这个脚本会构建产品（自动获取产品依赖的所有内容），为它构建安装工具，并测试这个安装工具。就这么简单，Bob 已经完成了他的工作。

从上面描述的两个场景中，可以看出使用一个自动化构建系统与手动组装产品之间的差别。Billy 犯了很多错误，而 Bob 通过自动构建避免了这些错误。Billy 忘记的步骤，Bob 的脚本都会为他自动完成，而且每一次做法都相同。

尽管很多人看到 Billy 工作到很晚，认为他更敬业，不过我们还是更愿意与 Bob 共事（或者让自己成为 Bob）。自动完成构建过程不仅可以更准确地完成各个步骤（更不容易出错），还能让你按时收工——就像 Bob 一样。

可以采用很多不同的方法来构建产品。你的步骤可能如下所示：

(1) 编译代码；
(2) 将编译的代码复制到安装工具目录；
(3) 移动第三方库（如数据库驱动程序和解析工具）的最新版本；
(4) 放入非代码文件（如 HTML 和图像）；
(5) 将帮助文件复制到安装工具目录；
(6) 构建安装工具。

手动完成构建或打包过程中的任一工作，都可能出现问题。关键在于，你是要选择先投入时间解决这个问题，还是打算浪费时间反复地手动完成任务，然后处理因为漏掉某个步骤或失误而导致的种种不可避免的错误。前者绝对是明智的选择，即在项目的初期投入时间。后者将是一个黑洞，它会成为产品生命期中所有麻烦和问题的“万恶之源”。也就是说，如果地基建得不用心，要想盖一幢牢固的房子，你就必须下更大气力。

技巧 4

从第一天起就使用脚本构建

至少，要使用一个批处理文件或 Shell 脚本来完成构建（当然，还有更好的办法，稍后就会看到）。

“不过请等等!”你可能会争辩说，“为什么不能直接用我的IDE来完成这些步骤呢？”嗯，当然也可以，但是这样做会存在一些问题。

- 构建机不太可能也使用相同的 IDE（大多数 IDE 都很难甚至不可能采用批处理模式）。
- 你必须要求整个团队——包括专家和新手——都使用相同的 IDE。有时这尚能接受，但是很多情况下，这样做带来的问题远比它解决的问题还要多。
- 即使每一个人都使用同样的 IDE，也可能很难在开发环境中向每一个人传递变更情况（比如使用了一个新库，或者采用了不同的编译器设置）。

这些问题并非不能解决，但确实很棘手。正因如此，我们认为更容易的做法是将自动构建与 IDE 分开。重要的是，构建（尽管如此复杂）只需一个命令就可以启动。

如果无法做到仅用一个命令就能完成项目构建，团队的大部分人可能就懒

得去构建了。就像大多数任务一样，如果不能轻松地完成，没人会乐意去做。要检验一个开发项目是不是很棒，标志之一就是在每一个开发人员的计算机上都能很容易（而且一致）地构建这个项目。

尽管我们非常希望自动构建与IDE脱离，但事实往往并非如此。我们曾见过这样一个工作室，他们总是在一台特定机器上的特定的IDE中点击“构建”按钮来构建他们的产品。后来创建这个构建的开发人员离开了这家公司。剩下的员工没有人知道如果不使用这位离职开发人员的工作站和那个“神奇的构建按钮”该如何构建系统。我们指出代码应如何构建时，经理们都很震惊——他们根本对此一无所知。

要保证在工作室的每一个工作站上都能以完全相同的方式构建产品。如果你都不知道如何构建这个产品，那么很可能别人也不知道。

技巧 5

任何机器都可以作为构建机

必要时，只要使用相同的签入脚本，任何开发人员的机器都可以作为构建机。我们的目标是任何有构建条件的机器都应当能完成与构建机完全相同的构建（除了时间戳、IP地址或机器名等有所不同之外）。

构建脚本不必非常复杂。下面的例子显示了一个完整而且可用的Ant脚本，这个脚本非常简单明了，而且（除了XML尖括号）很容易看懂。

```
<?xml version="1.0" ?>
<project name="SimpleExample" default="doall">

        <property name="build.dir" value="./build" />
        <property name="dist.dir" value="./dist" />

        <target name="init">
                <mkdir dir="${build.dir}" />
                <mkdir dir="${dist.dir}" />
        </target>

        <target name="compile" depends="init">
                <javac srcdir="."
                           destdir="${build.dir}"/>
        </target>


        <target name="dist" depends="compile">
```

```
                <jar destfile="${dist.dir}/SimpleExample.jar"
                     compress="true">
                   <fileset dir="${build.dir}" />
                </jar>
        </target>


        <target name="doall" depends="dist">
        </target>

</project>
```

如何起步

如果你刚接手一个没有自动构建的新产品，应该立即采取以下步骤得到一个构建脚本：

(1) 让一个团队成员手动地构建系统，你做好记录。

(2) 定义各个构建步骤。

(3) 选择一个构建工具，同时也要准备其他几种可选工具，以备不时之需。

(4) 增量式建立每一个步骤的脚本，逐个删除手动操作。

(5) 在另一个工作站上运行脚本。这一步可以找出所有偶然出现在个别工作站上的代码。

(6) 现在让另一个团队成员尝试使用这个脚本，你不要提供任何帮助。

完成这些步骤之后，就可以得到对每一个人都适用的脚本。

我做得对吗

适当地使用你的手动构建系统，你就可以构建整个产品并保证：

- 只使用一个命令就可以完成构建；
- 会根据源代码管理系统（SCM）来构建；
- 可以在任何团队成员的工作站上构建；
- 没有任何外部环境要求（如特定的网盘）。

如果这些方面存在问题，就要重新审视你的构建过程。

警告信号

- 构建中包括手动步骤。
- 必须修改你的构建脚本才能在不同的机器上运行。
- 只有少数团队成员知道如何编辑构建脚本。

4 自动构建

不需要人照管的构建就是自动构建。不过，在实现一个自动构建之前，必须有一个只需一个命令就能运行的手动构建系统。如果没有这样一个手动构建系统，可以再返回去学习实践 3。你不能对一个不存在的过程实现自动化。

如果能够自动构建产品，应当多长时间构建一次呢？理想情况下，每次代码变更时都要重新构建。这样一来，一旦某次变更破坏了你的构建，你马上就能知道。为这个系统增加一组轻量级的烟雾测试（smoke test），这就相当于有了基本的功能保险措施。这种系统称为持续集成（Continuous Integration，CI）[①]。持续集成工具放在一个无开发人员介入的干净的构建机上，每次有人提交代码时就可以重新构建项目。由于每次提交代码时都会重新构建，所以一旦有编译错误出现就可以立即捕获，从而保证代码基是干净无误的。它还会运行测试套件来捕获功能错误。我们使用了一个开源的 CI 工具，名为 CruiseControl[②]，因为它有良好的支持，伸缩性好，而且还是免费的！

遗憾的是，大多数开发工作室（将近 70% [Cus03]）甚至连每日构建都做不到，更不用说 CI 系统了。有 CI 系统的工作室往往都是最棒的，能始终如一地生成较好的代码和较稳定的产品。你可能会有异议，认为能引入这种技术的工作室往往都很先进，当然会生产出更好的代码。我们可不同意你这种说法！

我们认为，这类工作室的优势在于，他们一直有一个“虚拟构建监视器”能够立即捕获提交的每一行不好的代码。它能标志出无法编译的代码，还可以捕获你忘记增加的新文件或者修改过的现有文件。自动构建系统非常擅长捕获我们人类容易遗漏的细节。

技巧 6

持续构建

除了注意“能不能编译”，我们还可以进一步询问“能不能运行”。利用一个精心选择的测试套件，可以对基本功能重新测试，而不再重新引入 bug（避

① http://martinfowler.com/articles/continuousIntegration.html。

② CruiseControl 是一个开源的持续集成系统，由 SourceForge.net（http://cruisecontrol.sourceforge.net/）维护。

免 bug 回归）。利用这个系统，开发人员可以把时间用来增加特性，而不是修正编译问题，也不是反复修正同样的 bug。手动测试无法为开发人员提供这种便利。对每个代码变更立即做出反馈可以很快捕获到问题，因此问题可以很快地得到修正。这也是具备 CI 系统的工作室总是超越其他工作室的一个主要原因。


小乔爱问……

什么是 bug 回归?

bug 回归（bug Regression）是指你原先已经修正的一个问题又重新潜入代码。如果你在问题修正之后意外地将不好的代码又放回到 SCM，或者重新引入同样的错误，就会发生这个问题。bug 回归确实是一个很让人头疼的问题。

如果为修正的每一个bug编写一个测试，并在CI系统中运行，在签入有问题的代码时系统就会捕获到 bug 回归。这种策略可以有效地防止 bug 回归。

要让你的工作室也拥有一个 CI 系统！没有你想象的那么困难，而且它确实会带来巨大的好处。

SAS（世界上最大的私营软件公司，碰巧我们也为这家公司工作）共有超过 5 000 000 行代码由一个 CI 系统监视。实际上，很多个分公司都达到了这个覆盖率，所以有数倍于 5 000 000 行的代码得到监视！（见[Cla04]中我们的故事）。既然 CI 能适应这么大的规模，它在你的工作室里也肯定适用。

技巧 7

持续测试

CI 系统真的如此重要吗？可以告诉你，在我们参与过工作的一家规模很大的公司，有一个关键项目的构建失败导致了一连串的问题，以至于公司近 150 个其他项目都在那晚构建时失败。CI 系统在下午较早时候捕捉到这个错误，但是系统还只是 beta 测试版，没有人监视，所以那天晚上近 150 个项目都未能构建。这个小故障影响了分布在各大洲的众多开发团队。在这个事故之后不久，管理层加速在全公司推行 CI 系统，以避免此类错误再次发生。

表示

一旦决定建立一个 CI 系统，就要好好考虑你希望如何表示结果。每个 CI 系统都允许将结果发布到 HTML（见图 2-4）。大多数 CI 系统还允许发送 email（见实践 14）。不过，这只是冰山一角。只需稍做调整，你的系统就可以采用更有趣的方式显示结果。

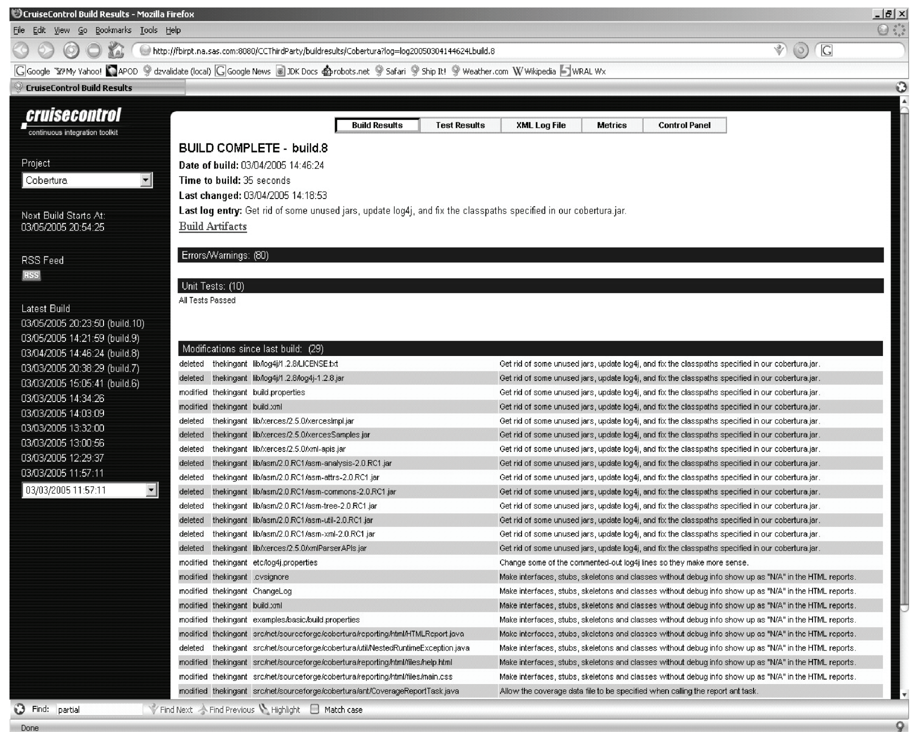


图 2-4 CRUISECONTROL 状态报告页面

需要启用构建系统的 RSS 提要。这会把构建信息以推送方式推至订阅人，而不是不断发送 email 骚扰他们。

增加熔岩灯！大多数构建系统可以使用 X10 模块[①]来驱动所有的可视化设备。有些人使用熔岩灯[②]，有些人更喜欢磨砂玻璃球[③]，你可以驱动你喜欢的任

① 利用 X10 模块可以控制计算机的电力开关。人们用它来驱动各种各样的设备，从电灯到熔岩灯。有关内容请访问 http://x10.com。

② 如 Mike Clark（http://www.pragmaticprogrammer.com/pa/pa.html）。

③ Michael Swanson 的博客是 http://blogs.msdn.com/mswanson/articles/169058.aspx。

何设备。试着将各种通知方式都尝试一下吧。

如何起步

你必须先有一个很好的构建系统，才能进一步建立一个自动构建系统。

(1) 选择一个要使用的自动构建系统。不要自行编写构建系统①。

(2) 用一个“干净”的机器来运行构建系统。

(3) 安装自动构建系统，根据你的环境进行配置。记录每一个安装步骤。

就这么简单！建立这种系统非常容易，而且通常只需几个月，投资回报（管理层称为 ROI）就能达到收支平衡点。如果管理层看不出系统的好处，可以先在你自己的机器上运行这个系统。从来没有用过 CI 系统的人通常需要一个现场演示才能认可这个概念的强大作用②。

如果一棵树在森林里倒了……③

如果不查看结果，即使是世界上最好的自动构建系统也没有任何用处。大多数系统会自动发送 email，要充分利用这个特性！我们就曾经见过很多人刚开始对这个特性很抵触，不过一旦团队适应，就会非常乐于接受这些 email（当然也从中得到很大好处）。我们会越来越依赖这些通知，甚至要等到收到“一切正常”的消息才会安心下班。

我做得对吗

如果你在使用一个自动构建系统（或 CI 系统），你就已经远远超前于大多数软件团队了。不过还有几个问题需要问问你自己。

- 系统中有测试吗？毕竟，如果只能编译而不能运行，那根本没有意义。
- 有人注意这个系统吗？通知功能打开了吗？
- 构建问题能很快得到修正，还是会数日都无人问津？

---

① 我们见过很多人想建立一个简单的通知系统，但是既然可以免费使用一个已经充分测试的企业级 CI 系统，为什么还要那么做呢？你增加的特性或报告可能还不及一个完备项目的一半。参见附录 D，其中列出了一些很好的 CI 产品。毕竟，重复创造不是一种务实的做法。

② 要想快速了解，可以查看 Mike Clark 的 CruiseControl 使用视频，见 http://media.pragprog.com/movies/auto/CruiseControl_MikeClark.html。

③ 这句话源于一句谚语：“如果一棵树在森林里倒了，却没有人在场听到，那么它发出的声响又算什么？”——译者注

- 构建能在一个合理的时间内完成吗？还是需要花费很长时间？

如果所有这些问题的回答都是肯定的，说明你的团队可以把时间用来增加特性，而不是跟踪究竟哪一行代码破坏了上半年某个时间增加的特性 X 了！

警告信号

- 自动构建系统频繁崩溃。
- 你的团队对失败的构建置之不理。
- 构建停止运行，但没有人注意到。

5 跟踪问题

问题列表的概念非常简单：有人报告一个 bug 或其他重要问题时，你肯定想留下这个报告的一个副本以免忘记。很简单，是不是？［如果我们谈论 bug，开发人员通常很不高兴，所以这里使用一个更柔和也更没有歧视之嫌的术语：问题（issue）。］

不过，跟踪问题不只是描述问题本身。跟踪并有效地交流问题的细节不是那么容易就能做到的。你需要知道以下信息：

- 哪个版本的产品存在这个问题？
- 哪个客户遇到这个问题？
- 这个问题有多严重？
- 这个问题可以在公司内部再生吗？（由谁再生？如果你自己不能再生这个问题，就可以向他们寻求帮助。）
- 客户的环境是怎样的（使用什么操作系统、数据库，等等）？
- 这个问题最早出现在产品的哪个版本中？
- 这个问题在产品的哪个版本中得到修正？
- 谁修正了这个问题？
- 由谁验证了该修正？

经过一段时间后，你会注意到白板或 3 × 5 的小卡片已经不够用了，项目的复杂性会飞速增长。这些媒介当然可以存放信息，但是无法搜索，也不能发布到网页上提供给技术支持团队。另外，还不能很好地扩展或共享。因此，需要把这个信息保存在数据库中。

当有客户打电话来指出问题时，技术支持可以利用数据库很快查找到这个问题。他们能做好准备，告诉客户你已经知道这些问题以及这些问题是否会在下一个版本中得到修正（见图 2-5 给出的例子）。

你要创建一个覆盖整个产品（甚至整个公司）的长期记忆，否则公司会得阿兹海默病（Alzheimer's disease），也就是老年痴呆症：经历一番艰苦努力后，却发现这个问题以前已经发现过，这不仅浪费时间，让你的客户备受困扰，也让测试人员和开发人员很头疼。[①]

① 据说，只要你告诉客户问题已经在已知问题列表中，不论这是什么问题，你都能得到原谅。不过如果你自己的 bug 要由客户来告诉你，他们肯定不会原谅你。

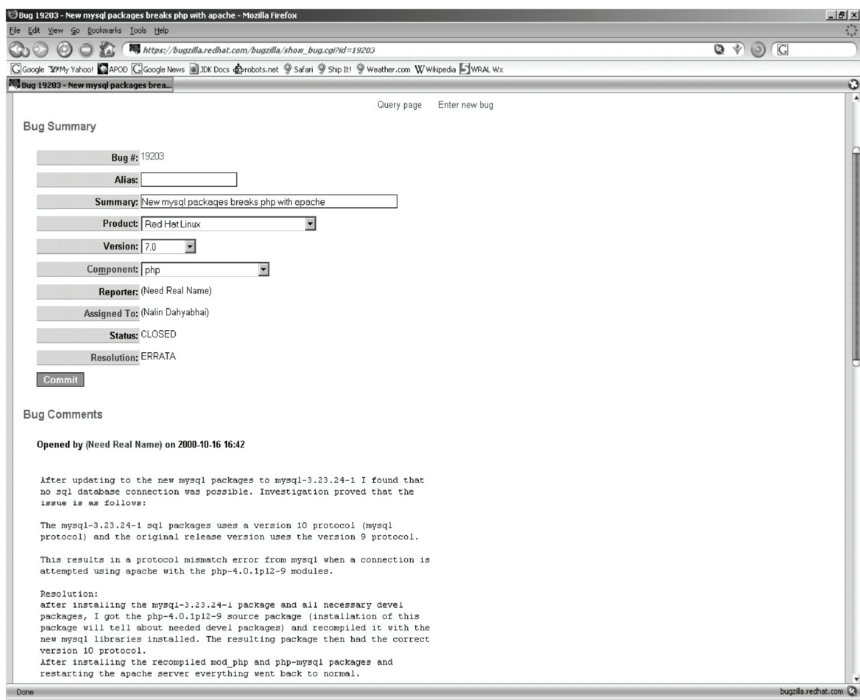


图 2-5　示例问题跟踪系统输入屏截图

技巧 8

避免集体失忆

一个好的问题跟踪系统会为给定的产品生成活动报告。遇到多少个问题？多少个问题得到解决，花费了多长时间？等等。这种报告可以用来找出项目中的问题地带。（研究表明，bug、麻烦和“问题”往往会聚集在一起——一般不会均匀分布。）

跟踪系统不仅可以为产品生成已知问题列表，还可以为开发人员生成任务列表。如果没有人修正问题，就没有必要浪费时间来查找问题。

来看 Harry 和 Lloyd 的例子，他们是一家小型软件创业公司的开发人员，非常能干。一开始他们是通过将即时贴粘在办公桌前面的墙上来跟踪每一个问题的。这些小纸片是他们跟踪 bug 的唯一途径，他们向我们保证这个系统很可靠。我们还是坚持使用一个标准的问题跟踪包。最初他们有些抵触，认为我们在“修正本来没问题的系统”，但是最终还是迁就了我们，迁移到这个新系统。

这家公司有一个很大的客户，是他们的忠实客户。该客户有一天打来电话，希望知道什么时候问题 X 能够得到修正。他们说早在两个版本前就已经报告了这个问题，现在已经等得不耐烦了，想知道为什么我们一直未予理会。Harry 和 Lloyd 坚持说他们从来没有听说过这个问题，这让客户很不满。最后我们修正了这个问题，客户气消了一点，但是对我们再也没能像原先那么“贴心”了。

六个月后，一次搬动 Harry 的办公桌时，我们发现一张即时贴掉在地上。上面用红笔写着大大的“非常重要！千万不能忘记！”几个字，后面写着那个客户的问题。原来是这张小纸片掉到了地上，“眼不见，心不烦”，他们就把这个问题忘到了九霄云外。使用问题跟踪系统，你就能为自己提供一个 Harry 和 Lloyd 从来不曾有的安全网。

问题跟踪系统就是一个簿记明细。你需要它记录你做的工作，修正了的和没有修正的问题，以及计划修正的问题。白板、索引卡或活页本或许可以应付几个月，但不是长久之计，而且肯定无法适应企业的具体要求。

如何起步

如果你现在还没有一个问题跟踪系统，那千万不能再等了。不要等到迁移完每一个遇到的问题之后才考虑迁移到这个系统。这是一个宏伟的目标，但是不要干等着手动系统“干净”了才使用自动系统。只要可以，就立即开始使用。把新问题输入到你的新系统中，过一段时间就会积累大量的信息，成为至关重要的资源。如果你能有时间为新系统填入之前的问题跟踪信息，那当然太棒了！不过不要把这作为启用问题跟踪系统的一个前提要求。

(1) 选择一个问题跟踪系统（见附录 E）。
(2) 为自己建立一个测试系统，学习如何使用。
(3) 为内部用户生成一个简短（只有一页）的快速入门指南。
(4) 着手将所有新问题都放在这个系统中。
(5) 在时间允许的情况下，将原来存在的问题转移到这个新系统下。

我做得对吗

用任何类型的系统跟踪问题，都很好。关键是系统中是否存储了足够的信息，以及内部人是否都在使用这个系统。可以用以下问题来衡量自己是否成功。

- 你能生成一个最高优先级的未解决问题列表吗？第二优先级的问题呢？
- 你能生成上个星期修正的问题列表吗？
- 你的系统能查阅修正问题的代码吗？
- 你的技术领导人是否使用这个系统为开发团队生成任务列表？
- 你的技术支持团队是否知道如何从这个系统中得到信息？
- 系统能不能通知“相关方面”，使技术支持人员（和其他人）可以了解问题何时得到修正？

警告信号

- 没有人使用系统。
- 系统中存在太多小问题。
- 把发现多少问题作为度量标准来评价团队成员的绩效[①]。

① 这是一种滑坡效应（slippery slope）。人们往往会认为报告或修正更多问题的人生产效率更高，但是用这些标准来衡量绩效是不合适的。如果你根据输入的问题数来作为奖惩标准，你的系统很快会塞满吹毛求疵的不重要的 bug。

6 跟踪特性

产品的新特性是指另外增加的功能，可以让产品做到之前不能做到的事情。例如，让你的产品与另一个供应商的数据库交互就是一个特性。另一方面，让产品在所支持的数据库上正确地工作则是修正一个 bug。这二者是有区别的，要区别对待。

不是一个 bug，这是一个特性！

bug（也就是问题）简单定义为：你希望软件做到但它没有做到的某些方面。这个定义涉及了数据丢失、系统崩溃以及简单的拼写错误等各种问题。

而特性则是对一个现有产品的改进。千万不要让一个过分热心的销售或市场团队把改进归为bug，努力将他们最喜欢的特性上升到一个更高优先级。如果一个特性很重要，那么可以指定一个合适的优先级。把它重新划归为bug只会“污染”问题跟踪系统。

有很多理由让我们相信，应当正确地跟踪特性请求。这使你可以评估客户对新特性的需求，并为以后的产品版本生成任务列表。知道特性是谁请求的，这对于指定优先级也很重要。当知道一个小客户请求了一个特性，另外有六个大客户请求了另一个特性，你肯定能正确地指定特性列表优先级。这两个特性可以同时增加，不过重要的是找出一种方法来确定工作的顺序。

如果没有进行跟踪，你很可能会忘记其中一些特性，这会让请求这些特性的客户认为你根本不在乎他们，从而发出一个不好的消息。

跟踪特性的方法与跟踪问题列表相同。需要维护一个统一的特性请求列表。为特性指定优先级，并对研究或增加这个特性可能需要的时间做一个基本估算。还可以在白板上保留最高优先级的特性列表，让大家都能一目了然。

当心特性蔓延

很多产品完成 90%后，就无法更进一步，似乎永远都完成不了。总有一些新特性需要完成。不论开发团队做了多少工作，总会出于某

> 种原因无法交付产品。这些产品就遭遇了一种称为“特性蔓延”（feature creep）的问题。简单地讲，特性蔓延就是特性一直在增加。现有的特性不断改进但永远不能达到完美。最终你发现这是在“磨白”[1]。需要一种方法来决定何时停止打磨，真正交付产品。要解决这个问题，一定要确保每个特性和 bug 都关联有一个确定的优先级。将所有的工作都排定出优先级，这样管理层可以在优先级列表上更容易地做出选择，决定哪些交付而哪些不交付。这不再是一个随意或带有个人偏好的决定，而是基于这些变更对技术领导人、销售团队以及最终客户的重要性来做决定。
>
> 同样，如果一个任务不在指定有优先级的列表中，就不要做任何处理（见实践 10）。

问题和特性列表通常都放在同一个跟踪系统中。不过一定要有一种清楚明确的方式来区分这两个列表。如果无法为产品生成单独的重要特性请求列表和重要 bug 列表，就会遇到问题。如果必须使用两个单独的产品，当然也可以，但是这不是必要的。大多数工具都可以做出区分。

俗话说：“如果你没有把它写下来，它就永远没有发生过。”比写下来更好的做法是，让计算机为你跟踪记录。如果没有这样做，事情就会像从来没有发生过一样……

如何起步

类似于“跟踪问题”（见实践 5）中的“如何起步”，这里的做法基本相同。

我做得对吗

如果以下成立，说明你做得很好。

- 要生成下一个版本的特性列表时，你首先会使用这个系统查找相关信息。
- 你会定期在系统中记录新的产品想法。
- 拒绝很多提交的特性。否则，在输入之前就将其剔除。

---

[1] “磨白”（sanding through the finish）是一个木工术语，指用砂纸打磨家具使外表更光滑，但却打磨得过度，以至于磨白，开始危及木材本身。这是新手常犯的一个错误。

- 可以从这个系统运行一个报告来生成产品前一个版本的“新特性”列表。
- 干系人[①]可以很容易地检查特性状态，并因为与他们的期望一致而感到心满意足。

警告信号

- 没有增加新特性。
- 所有特性都是最高优先级。
- 增加了特性但从未实现。
- 增加了不相关的特性。

① 解释见第 66 页。——译者注

7 使用自动化测试框架

自动化测试框架（testing harness）是用来创建和运行测试的工具或软件工具包。除了自动化测试，还可以手工编写独立的测试（或者更简单，根本没有测试）。

如果你的测试不是自动的，就无法用一个脚本来运行这些测试。你需要一个人来运行你的测试套件，这就需要花费时间和金钱。而且人们每次做事情总会稍有差别。交互式测试对于测试工作很有意义，好的自动测试套件也一样。好的测试套件就像优秀的测试人员或开发人员一样，价值非比寻常。它有助于你的产品有上佳表现，能够很快捕获问题，向开发人员迅速做出有关产品状态的反馈。在 CI 系统中运行一组好的自动测试能大大提高工作室的开发质量，没有比这种做法更好的了。

技巧 9

演练产品——自动测试

使用一个“现成的”测试框架有很多好处。例如，与你自己创建的测试框架相比，它的特性集会更为全面。如果你选择一个知名的框架，还可以找到很多辅助产品。例如，很多 XUnit 自动化测试框架都提供有支持产品，可以根据其输出格式生成报表。MetaCheck[①]就是一个很好的例子，这是一个开源工具，可以收集多个代码检查人员的输出并进行格式化。如果访问 SourceForge.net 并搜索 JUnit，你会发现有几十个项目已经对 JUnit 的基本功能做了扩展或改进。使用一个标准框架时，可以免费得到很多额外的好处。

相反，如果你的工作室没有一个标准的、兼容的测试框架，这意味着你采用了一系列不兼容的框架。（如果让三个开发人员和两个测试人员解决同一个问题，你可能会得到九种不同的解决方案。）

除非你希望团队中每一个人反复解决同样的问题，不停地重复劳动，否则就应当有一个所有人都可以使用的通用框架。换句话说，每个人都使用同样的框架才能最高效地促进工作。要选择一种灵活的轻量级框架，可以针对特定项目的需求适当调整。它不能太过严格，否则无法适用于多个项目。

① http://metacheck.sourceforge.net。

缺陷驱动测试

缺陷驱动测试（defect-driven testing）瞄准原来存在（或者现在仍然存在）缺陷的代码区域，并以此来指导创建测试。这是一个简单而有效的创建测试的方法，只查看现在产品中哪里有 bug。不用担心以往存在问题的区域，只需查看开发人员今天吃午饭时抱怨的问题。他们现在在跟踪什么代码？找出开发人员今天处理的代码，并用它来创建你的自动测试。

使用缺陷驱动测试可以快速构建一个测试套件，直接解决开发团队当前的问题。如果代码中某个区域变得不稳定，这种测试能很快使这个代码“免疫”，防止这些活动的缺陷反复出现。如果一个产品区域是稳定的，对测试覆盖率就不会有这样紧迫的需求。如果资源很充足，我们还是需要尽量保证自动测试100%的代码覆盖率，但是在现实中，往往需要使有限的资源发挥最大效用。

要确保自动化测试框架有一个命令行接口，以便通过一个外部脚本或工具来驱动。还应当能够处理单元测试、产品测试或集成测试。

技巧 10

使用通用、灵活的自动化测试框架

测试的类型有很多种，它们识别问题的类型也各不相同。

单元测试（unit test）用来测试单个的类或对象。这些测试是独立的，通常不需要运行其他类或对象。单元测试的唯一作用就是验证一个代码单元中逻辑操作的正确性。

功能测试（functional test）用来测试整个产品的操作或功能是否正确。这些测试可以针对整个产品，也可以针对产品中主要的子系统。功能测试会测试系统中的多个对象。

性能测试（performance test）测量产品或一个关键子系统的运行速度。如果没有这些测试，就无法知道一个代码变更对产品响应时间的影响（除非你很擅长掐秒表）。

负载测试（load test）模拟产品在很大负载情况下的表现，这些负载可能来

自大量客户，也可能来自一组任务繁重的用户，或者二者都有。同样，如果没有这种类型的测试，你将无法客观地说明代码基是得到改进还是有所退步。

烟雾测试（smoke test）是一种轻量级的测试，必须经过仔细编写才能测试产品的关键部分。它们运行得很快，但仍然能测试产品的重要部分。其基本思想是运行产品来看它是否“冒烟”，也就是说调用基本功能时是否会失败。烟雾测试非常适合与 CI 系统联用（见实践 4），因为它的速度很快。在产品的生命周期中，经常运行的烟雾测试可能会循环。烟雾测试套件主要针对活动的开发区域或已知的 bug。

集成测试（integration test）查看产品线的各个部分如何集成在一起。这可能涵盖多个产品，有时是你的产品，有时还有你使用的第三方产品。例如，产品使用的各种数据库就可以作为集成测试的一部分来测试。这些测试往往会超越产品边界。集成测试通常用来验证产品所依赖的组件的新版本，如数据库。如果你喜欢的数据库推出了一个新版本，你肯定想知道产品能否在这个新版本上运行。一组集成测试可以从功能一直测试到数据库，这组测试将为你回答有关功能的问题，还会让你迅速了解这些新组件的性能。

模拟客户测试（mock client testing）用来从客户的角度创建测试。模拟客户测试设法为产品展现常见的使用场景，确保产品满足最低功能要求。这种测试绝对能够在保证基本测试覆盖率的前提下，涵盖最常用的代码路径。请参见下面的说明。

模拟客户测试

模拟客户测试是我们提出的一个概念，正得到迅速普及。想想看你的产品将如何使用。它可能用于客户的应用（如果你开发的是一个产品），或者由底层代码使用（如果你在编写一个 API）。可以编写一个测试套件来模拟客户的行为。模拟客户测试像一位普通的客户一样使用产品（或子系统）。你会“模拟”一个客户，类似于“模拟对象”（Mock Object）[①]模拟一个服务器或应用资源。这些通常归入集成测试，因为可以对实际系统运行这些测试。这些测试可以与性能测试和负载测试“串连”在一起，也可以单独用作烟雾测试。这个概念相当灵活，而且非常强大！如果有一组用户场景，那么可以把它们放

① 参见 http://www.mockobjects.com/Faq.html。

在一个模拟客户测试中，验证确实能运行（重构代码基之后还能继续运行）。

使用模拟客户测试，可以执行系统中最重要的代码路径，也就是运行代码会调用的那些路径。这可能无法提供其他测试方法所能达到的代码覆盖率，但是能覆盖最重要的代码。对于测试资源紧缺的工作室，模拟客户测试尤其重要。

在各种类型的测试中，模拟客户测试可以为你的投入给出最好的回报。如果你负责的一个项目没有测试，就可以从模拟客户测试开始，以后再增加其他类型的测试。可以结合模拟客户测试和缺陷驱动测试，这种策略可以确保测试以最高效的方式覆盖最需要关注的代码。

不要被不好的测试框架捆住手脚。一定要确保有一个非常灵活的产品。明天要测试的东西可能与今天完全不同。如果你有一个灵活的产品，就可以使用这个产品继续测试。如果框架过分特定或过于严格，就不得不另找工具包从头学起。时间是最宝贵的资源，不要把时间浪费在学习不可重用的工具上。

在一家规模很大的公司里，我们看到一个缺陷驱动测试和模拟客户测试相结合的典范。吃午饭时，我们听到有开发人员在抱怨，他们产品内部的一段代码在过去一周已经出了三次问题。这个问题太严重了，足以导致整个产品全盘失败，而团队已经花了近一天的时间来查找这个问题。第一次修正这个问题时，却引入了另一个与之相关的问题。找出这第二个问题又花了半天时间。接下来，修正第二个问题时再次引入了第一个问题。这一次找到这个问题倒是只花了几个小时。每一次整个软件团队都全员参与，所有工作都停顿下来。他们无法做任何其他工作。我们做了一个粗略的计算，发现这个团队为处理这个问题已经浪费了两个人月。

得知这种情况后，我们让参与这个工作的测试人员调整了方向。让他们从上层代码的角度（即从底层问题代码的一个客户的角度）创建测试，并编写测试专门“曝光”上周的 bug。一个测试人员只花了半天时间就完成了这个基本测试，示例代码由已经花了大量时间调试并已明确定义问题的开发人员提供。

然后我们把这个测试放入一个 CI 系统，每次开发人员增加或修改代码时它

都会运行测试。接下来一周有人两次重新引入了有缺陷的老版本代码。由于 CI 系统运行了这些新测试，它在新代码签入后半个小时内就捕获到问题。问题在一个小时内得到了修正和验证（通过自动测试）。如果没有 CI 系统和这个新测试，这个问题就会重新潜伏到产品中，再次导致同样的问题。

这些测试是从客户（上层代码）的角度编写的，这正是模拟客户测试的基本原则。另外测试功能由最近出现的 bug 标识——这正是缺陷驱动测试的工作原理。通过应用这两大原则，你就可以利用现有的测试资源，并成倍放大其作用。

下面是 JUnit 测试代码的一个例子（如图 2-6 所示，GUI 显示测试已通过）。

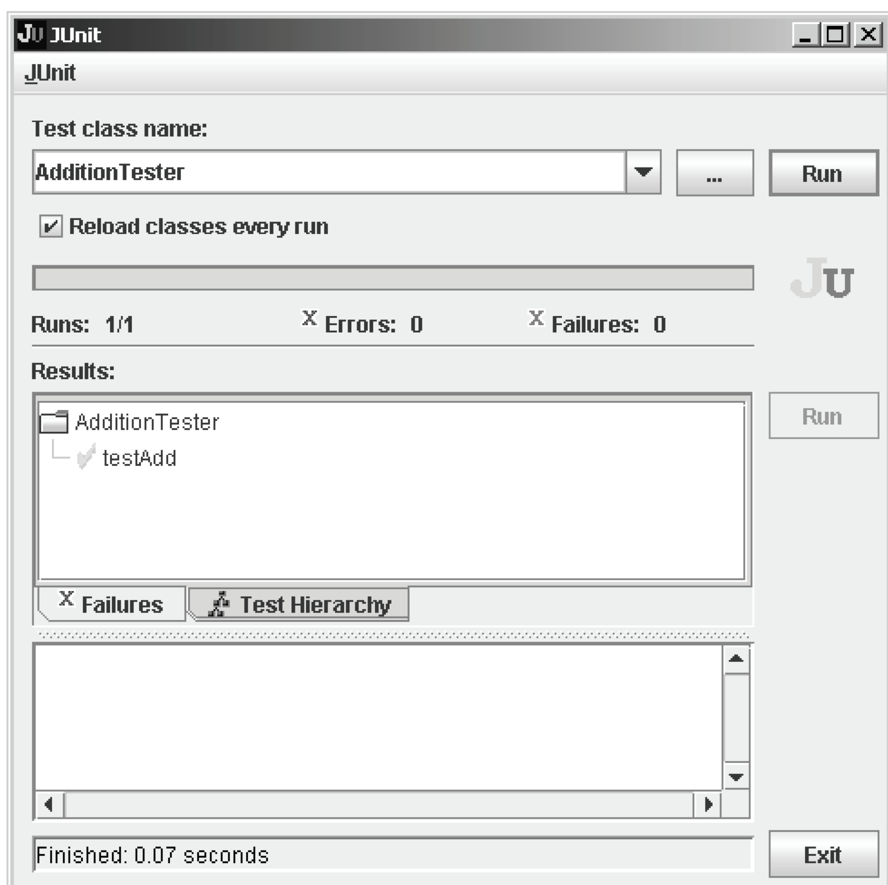


图 2-6　JUnit 控制台，显示所有测试已经通过

```
import junit.framework.*;
public class AdditionTester extends TestCase {
    public void testAdd() {
        assertEquals(5, 2 + 3);
    }

    public static void main (String[] args) {
        junit.swingui.TestRunner.run(AdditionTester.class);
    }
}
```

一个好的测试框架会对开发工作产生难以置信的影响。如果有效地使用，这会是一个强大的工具。一定要投入必要的时间找到一个适合你的具体环境的框架，然后学习如何有效地加以使用。

如何起步

如果你的团队根本没有测试，就要与你的技术领导人（或经理）谈一谈了。你可以在你负责的领域中加入测试，来展示测试的好处，但是要想得到最大的收益，还是需要整个团队都参与。

如果你的团队已经在写测试，要看看他们在使用什么框架。很多开发人员会编写他们自己的框架——请不要这么做！不过，如果工作室里有人采用了一个标准框架或工具，要了解这是什么框架，为什么会选择这个框架，充分利用他们的经验和专业技能。

一旦选择了一个框架，就要使用缺陷驱动策略为代码创建测试。强烈建议你创建模拟客户测试，并在一个 CI 系统中运行所有测试。CI 系统是确保定期运行测试的最佳方法。

(1) 选择一个测试工具或工具包。

(2) 开始为问题领域加入测试。

(3) 确保测试在一个自动构建系统中运行（见实践 4）。

不太清楚历史

如果代码中存在历史问题，并不一定意味着开发人员水平低。代码存在这些问题可能有很多原因。也许代码非常复杂，或者同一个领域有多个开发人员，他们的工作发生冲突。也可能是一名新来的开发人员接手同事的工作，刚刚转向这个新领域。

> 不要认为代码有大量 bug 就是不好的开发人员编写的。这时应当假设要解决的问题很困难。一旦认识到这种复杂性，你就能创建很有效的测试。

我做得对吗

回答下面的问题，从你的答案就能清楚地看出你做得对不对。

- 你的测试套件有效吗？测试能捕获 bug 吗？
- 你的代码覆盖率[①]是多少？会不会随时间增加？
- 你测试的产品稳定吗？
- 测试会自动运行吗？
- 测试能不能告诉你是否通过？（如果必须手动地确定测试是否通过，就不是自动测试。）
- 工作室里每一个人都有能力增加测试吗？（如果不能，说明框架可能太过复杂。）

警告信号

如果你的测试存在以下问题，就要再斟酌一下你的策略。

- 没有运行。
- 从来没有捕获任何问题。
- 运行花费的时间太长。
- 需要很大精力维护。

① 代码覆盖率（code coverage）是指测试执行的代码行数。

8 选择工具

对于我们讨论的各个类别，应该分别都有一个具体的工具，如果你没有做到这一点，就需要好好考虑一下了。例如，不要把构建系统与 IDE 捆绑在一起。利用当今最流行的 IDE，你几乎可以做任何事情。它们可以采用任何粒度处理你的代码。的确，这些通用的工具能做很多事情，但是哪一件都不能做到最好。

应当花些时间来研究哪些工具满足你的需要，而不是仓促间使用一个不能完全满足期望的临时工具。在使用的工具方面绝不能迁就。有大量商业或开源产品可供你选择，找出你需要的工具。你使用的每一个工具都应当最胜任相关工作，要在每个领域中寻求"最出类拔萃"的工具。

一定要保证你的工具使用一种开放的格式，如 XML 或纯文本。如果一个工具采用开放的格式读写，它就能与任何其他使用开放格式的工具"交流"（假设语义匹配，或者经过转换后能够匹配）。利用这种适应性，就可以把多个不同的工具串连在一起，构成一个完整的端到端系统。利用似乎毫无关联的工具之间的这种交互，可以实现让人难以置信的协同工作。

> **技巧 11**
>
> 工欲善其事，必先利其器

通过开源的 Java 构建工具 Maven 可以看到对这个概念的典范应用。Maven 使用 Ant 脚本和 Jelly（一种 Ant 脚本语言）封装了构建过程。测试会自动运行（Maven 可以驱动 JUnit 测试框架），然后 Maven 将测试结果转换为 HTML 报告。报告可以是原始的测试结果，也可以包括代码覆盖率甚至静态代码分析。可以使用 Maven 将这些工具串连起来，因为它们都使用开放格式作为输入和输出。Maven 只是为信息建立了一个管道。这里只是简要描述了 Maven 的工作，但从这么简单的描述中也能把握住重点。只要工具采用开放的格式读写，就可以把它们串连在一起。

作为一个反面例子，来看一种名为电子设计交换格式（Electronic Design Interchange Format，EDIF）的格式。几乎所有使用这种格式的早期程序都有一个很棒的导入工具，允许读入 EDIF 格式的数据。不过，所有程序都不允许导出 EDIF 数据，这样就能轻松地把你从其他竞争程序那里争取过来，而一旦一

个开发商有了你的数据，你就再也不可能脱身了。这个“特性”也使得你（或者任何第三方）几乎不可能编写工具来补充这些系统。这个缺陷完全破坏了EDIF的本来意图。要保证你使用的工具可以读写开放的格式。不要让开发商像通过EDIF一样“绑架”你。

技巧12

使用开放格式集成工具

9 何时结束试验

不要用一种只适合个体的小环境技术（niche）[①]或非核心技术来编写产品周期中的关键部分（如构建系统），特别是那种只有一个开发人员了解的技术。应当使用工作室里任何人都能配置和维护的技术。大杂烩式的科技游乐场（technology playground）固然很好，对于专业开发是必要的，但是在这里并不适用。试验应当避开关键路径[②]。

在一家小型创业公司里，构建脚本采用一种新语言编写：实际上这是编写脚本的开发人员的一个学习项目。更糟糕的是，这只是一种通用脚本语言，而不是一个构建系统。脚本包含 25 页意大利面条式代码[③]，几乎用到了各种晦涩的语言特性。无需多说，这个代码当然很令人费解。而且代码是硬编码的，相当依赖于开发人员的机器、特殊设置的网盘、软件组件的特定版本。这个程序简直是一团糟。

创建一个关键技术（如构建系统）时千万不要把它变成一个技术试验。要使用专门的构建工具来创建你的构建，而不是采用某个团队成员想学习的一种很酷的新技术。有很多不太重要的领域可以供你学习新技术。不要创建只能在一个机器上运行的自动化工具。不要由于硬编码而依赖网盘。把你需要的所有信息都放在 SCM 系统中，这样一来，网盘就变得不重要了。

例如，如果你在一个 Java 工作室工作，可以考虑使用 Ant 构建脚本。Ant 的本来目的就是为了构建 Java 程序，所以用 Ant 编写 Java 构建脚本要比使用 Python 之类的语言容易得多。Python 是一种很不错的语言，但是它对 Java 一无所知。选择构建系统时要充分利用你现有的专业技能。

遵循这个规则会带来很大好处——这会使工具的维护变得容易很多。工作室里的任何人都能使用这种技术，并做出调整。另外，通过强调经验，可以避免落入某种技术“陷阱”，因为有些技术看上去很适用于给定情况，但实际上不是。

---

① niche 来源于法语。法国人信奉天主教，在建造房屋时，常常在外墙上凿出一个不大的神龛，以供放圣母玛利亚。它虽然小，但边界清晰，洞里乾坤。这里是指适合个体的小环境。——译者注

② 项目的关键路径包含可能让项目速度减慢的所有部分。作为关键路径的组成部分，SCM 系统和构建脚本就是很好的例子。一旦这些部分出问题，所有其他部分也要停下来。

③“意大利面条式代码”一词通常用于描述捆绑在一起并且具有低内聚力的类和方法。——译者注

但是这实在太酷了！

不要拿核心技术作试验。如果你认为某种技术应该是这个规则的一个例外，就需要投入时间和资金让所有人都得到培训。不要只是单纯地引入新技术，自以为既然这个技术那么棒，所有人都会去学习。绝对不是这样。如果你希望人们学习某种技术，要通过反馈来确保他们确实在学习。要记住，这种反馈能反映老师的才能，而不代表学生是否愚蠢。

只有经验能够告诉你一种技术有什么缺点。

另外还有一个危险需要指出，不要让某个代码自动向导（或构建脚本）为你做你自己并不了解的事情。让一个工具帮你处理细节确实很好，但前提是你必须已经了解这些细节。如果你不了解，一旦出问题就会完全不知所措。“不要使用你不理解的向导代码。”[HT00]

技巧 13

使用熟悉的关键路径技术

不要把“活跃”与“进步”混为一谈。

►无名氏

第 3 章

实用项目技术

有些项目能生产高质量的软件，而且能够按时甚至提前交付；而有一些却延迟交付，超出预算，甚至干脆取消（不幸的是，这种情况还相当多）。有许多理论解释为什么会这样，也有很多方法可以采用。这些方法的一大基石就是改善协作。

除非你在自己家中开发代码，否则肯定要与其他人合作。即使你一个人在家完成一个项目，也是客户付钱让你做的。我们大多数人都在软件开发团队中工作。我们要与团队里的成员共事，共同完成我们开发的软件产品。

尽管良好的协作不能完全保证项目的成功，但如果协作不好，几乎可以肯定项目最终会失败。

这一章中我们会介绍一些协作技术，我们就是使用这些技术保证项目平稳而高效率地运行的。采用这些理念，你会变得越来越了解项目的状态和团队成员的进步。这些技术会让你的工作更轻松、更有成效，也更有趣。

并没有一个标准的项目改进技术清单列表。每个项目和每个团队都有所不同，而且不论是软件开发还是对团队动力的理解，现在人们还一直在不断探讨中。

在整个职业生涯中，你会不断发现需要学习的新技巧，也不断会有不再适用需要舍弃的习惯。这里介绍的技术是我们认为最有用的一些技术，具体包括：

- 任务清单（list）
- 技术领导人（tech lead）
- 每日例会（daily meeting）

- 代码审查（code review）
- 代码变更通知（code change notification）

你可能还记得，这些实践曾经出现在图 1-1 中，下面再次列出（见图 3-1）。

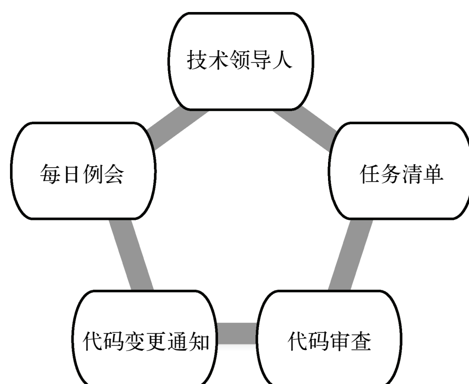


图 3-1　项目技术

10 按照任务清单工作

很多情况下我们会使用一个计划任务清单（to-do list）来跟踪我们的工作。任务清单（list）可以规范计划任务的概念，从而可以在团队环境中使用。

过去，完成一些较小的项目时，我们会使用一些标准笔记簿和记事本来跟踪记录工作。任务清单最初只是我们个人的计划任务清单，记录我们不希望忘记的事情。当我们升迁到领导职位时，使用的任务清单就会逐步包括整个团队的相关事项，光写在纸上就不能满足需要了。把任务清单向其他团队成员展示多次后，你就会开始寻求其他途径来记录和共享任务清单。对于小团体来说，白板就很不错，特别是在有许多隔间的大房子里（比如说创业公司？）。如今我们更愿意使用网页或 Wiki。有些人使用电子表格[①]来记录任务清单。所有人都可以访问网页，而且也很容易编辑。

可以利用任务清单来安排你每天和每周的日程。利用任务清单可以安排你的工作顺序，还可以安排整个团队的工作顺序。（有点分形的意思！）如果你陷入困境、头昏脑胀、四处救火，可以回过头来查看任务清单，重新把握重点。如果你被一个极其棘手的问题卡住，需要暂时回避这个问题，任务清单会告诉你先做哪些容易的事项。这样可以确保你总在处理最重要的任务，而不是表面上最显眼的任务。

为什么需要任务清单

你是不是经常参与这样的项目：每个人都忙忙碌碌，可是产品永远都无法完成？总是遗漏一些重要的特性，团队成员都在等待那些未能完成的特性，开发人员被“卡住”，不知道下一步该做什么。

应当有人把所有特性写下来，按优先级排序，使团队成员可以从任务清单最前面得到他们的下一项任务。这样一来，你就为团队建立了一个组织中心，而且不存在大多数重量级过程可能带来的开销。

有了这个任务清单，团队成员就永远不会无事可做。完成了当前任务后，他们可以查看任务清单，从高优先级的任务中选择一个特性，继续投入工作。

① Joel Spolsky 使用一个 Excel 电子表格来记录他的工作清单，而且认为你也应该使用这样简单的工具，对此他有很充分的理由。有关内容参见 http://www.joelonsoftware.com/articles/fog0000000245.html。

开发人员选择下一个任务时，总能选到对他们来说最有意思的工作，而技术领导人可以确信优先级最高的特性正在开发当中，或者是正处于任务清单中下一个要完成的位置。

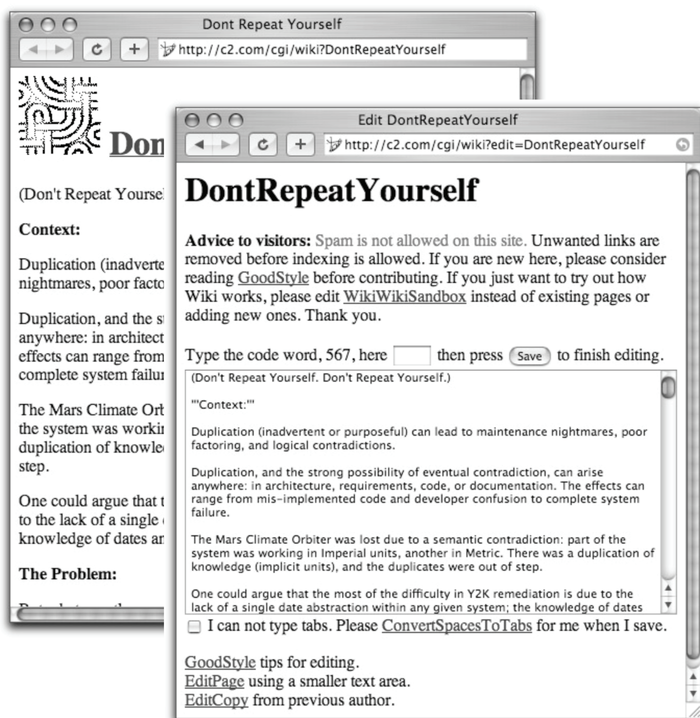
小乔爱问……

什么是 Wiki?

Wiki 是人们创建网页的一种简单方法。Wiki 使用一种简单的标记而不是用 HTML 编写，所以任何人都可以增加内容。Wiki 的设计就是为了鼓励团队协作。

Wiki 页面最初看起来与其他网页很类似，只不过页面最下面有一个“edit this page”（编辑这个页面）链接。点击这个链接会把当前页面的内容放在一个简单的 HTML 文本编辑框中，可以很容易地对页面进行修改。

更多相关信息请访问 http://www.wiki.org。



由于开发人员技术水平不一，技术领导人会在必要时做些调整，不过一般来讲，优先级最高的任务完成之前，不应去处理优先级次之的任务。

在团队真正投入时间增加特性之前，任务清单（作为一个团队工具）可以为管理层和客户提供一个具体的实物，供他们查看并对产品做出评价。你可能花了一个星期的时间来增加一个特性，而后却要将它去除，相比之下，尽早去掉这个特性肯定更划算！你刚完成一个产品，可是客户说："这个产品还不错，不过如果你增加了特性 A 而不是 X、Y 和 Z 的话，我肯定会更满意。"你是不是经常遇到这种情况？使用任务清单，你就相当于创建了一个轻量级的文档，可以在开发周期早期展示给大家。

任务清单还会为团队提供充分的敏捷性。它确保你已经根据需要将产品分解为特性，并将特性分解为列表项，从而保证你已经提前做了一些基本的设计工作。另外，由于产品已经分解为特性，所以可以根据需要去除或者增加特性。

大多数公司都曾经遇到过这种情形，一两个人突然闯进来，责问为什么特性 X 还没有完成，或者为什么还不处理那个特性（在他们看来，他们关心的特性就最重要的任务了）。如果你有一组明确的任务，而且指定了优先级，就可以向他们展示你正在开发哪些特性，解释为什么这些特性对于核心产品来说更为重要。这个解释往往足以让他们满意，表明你确实在做有用的工作。

这样一个已指定优先级的完备的特性列表将为你的团队、管理层以及其他依赖于你的工作的团队建立信心。如果任务清单包含很多深入的工作，说明你已经提前考虑并对后面的步骤做了计划。

技巧 14

按照任务清单工作

如何使用任务清单

可以对你自己的工作使用任务清单，也可以对整个团队使用任务清单。这两种方法都很容易，也很有效。这两种方式我们都将采用。

任务清单作为组织工具

多年来我们在工作中有过各种各样不同的角色，不过我们很少“只戴一顶帽子”[①]。一般来讲，与大多数人一样，我们也有很多不同的工作要做，但是具体完成这些工作的时间却很少。我们计划处理所有这些工作时，往往会把时间平摊开，由于任务太多，每项任务平摊的时间就很少，以至于最终根本无法真正完成工作。

我们的解决办法是把任务清单用作个人的组织工具。即使总是需要对整个团队使用团队列表，为我们自己必须做的工作建立单独的列表还是会很有帮助。每天早晨建立一个列表，确定需要完成的工作，并确定工作的优先级。一天结束时，再看你确实完成了哪些工作（或者没有完成哪些工作）。看看究竟是由于你过于乐观地考虑原以为能完成的工作，还是因为一天来总是因为其他事情而分心，而导致未能完成所有工作。关于如何使用任务清单来确定个人的工作列表，更全面的讨论可以参考 Stephen Covey 的《高效能人士的七个习惯》。

着手建立自己的任务清单很容易。首先，创建一个列表，其中包含在处理的（或没有完成的）所有任务。然后，与你的技术领导人一起，为每个任务指定一个优先级。最后，为每一项任务设定一个估计时间。一开始不用担心这些估计时间是否完美，以后可以逐步改进（参看图 3-2）。

如果你的产品已经明确定义，那么让团队按照任务清单工作并不难。这实际上是一个很好的团队活动，可以帮助整个团队了解项目的总体方向。

(1) 把将要为项目增加的各个特性写在一个白板上。这需要花一点时间，通常可能需要好几块白板。

(2) 为各个特性指定优先级。一定要让适当的干系人（管理层、客户等）加入这个过程。能让整个团队参与就太理想了，不过如果有些团队成员过于自负，只包括技术领导人和干系人可能会更顺利一些。

(3) 重写所有特性，按优先级排序。

(4) 为各项特性附加估计时间。

① 指只有一种思路。——译者注

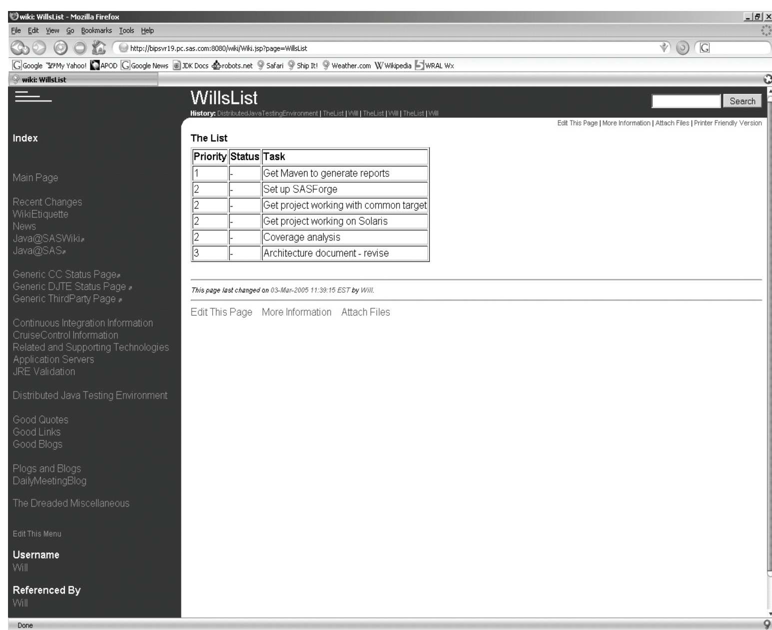


图 3-2　内部网页上显示的任务清单

在当前优先级最高的工作完成之前，任何人都不能处理下一个优先级的任务。这样可以确保所有第一优先级的任务得到处理后，才会处理第二优先级的任务。

可以看到，使用任务清单相当容易。不过，任务清单要真正做到有效，必须遵循一些原则。它必须包括以下所有特点：

- 可以公开获得
- 已指定优先级
- 有时间表
- 活跃
- 可测量
- 有针对性

接下来我们会了解这些原则的含义，以及它对我们以及团队意味着什么。



小乔爱问……

什么是 RSS?

不同的人会对 RSS 有不同的解释，可能是“Rich Site Summary”（丰富站点摘要）或者“Really Simple Syndication”（真正简单聚合），也可能是“RDF Site Summary”（RDF 站点摘要）。它实际上是一种分享内容变更的方法，在有动态内容的网站上非常流行（如新闻网站或构建状态）。一个网站使用 RSS 分享变更称为一个 RSS 提要（RSS feed）。RSS 提要就是一个列出了变更或新内容的 XML 文件。

RSS 阅读器（见图 3-3）是一个程序，可以检查你订阅的所有 RSS 提要并为你显示新内容。RSS 提要由 Web 服务器提供，RSS 阅读器只需要查看网站上的一个文件，并显示变更。

RSS 阅读器非常方便，可以采用一种摘要格式获取新闻。RSS 阅读器会一直收集新闻，供你阅读。

可以公开获得

团队的任务清单必须可以公开获得。一个秘密的任务清单对协作没有任何帮助。要把任务清单放在你的白板上或者放在网站上，为它建立一个 RSS 提要，不然至少要让人们很容易很明了地读到。把任务清单一直放在面前，这有助于保证工作重点。你能很容易地快速查看还有哪些未完成的工作——特别是在忙乱的一天中感到灰心或分心时，这会很有帮助。要保证任务清单可以公开获得，这样还能帮助经理了解进展情况。

已指定优先级

任务清单必须已经指定优先级。要区别产品中不同类型的特性——必要特性、可取特性和无用特性——这非常重要。在对任务清单指定优先级时必须有所区别，否则不分轻重缓急最后只会浪费你的时间。通常会有一组核心任务必须在产品交付前完成，这些就是优先级最高的特性。例如，这可能包括登录屏幕、安装程序和工作数据库。没有这些特性就无法交付你的产品。然而为“关于”对话框设置一个改进的新背景色则可以认为是一个无用特性。

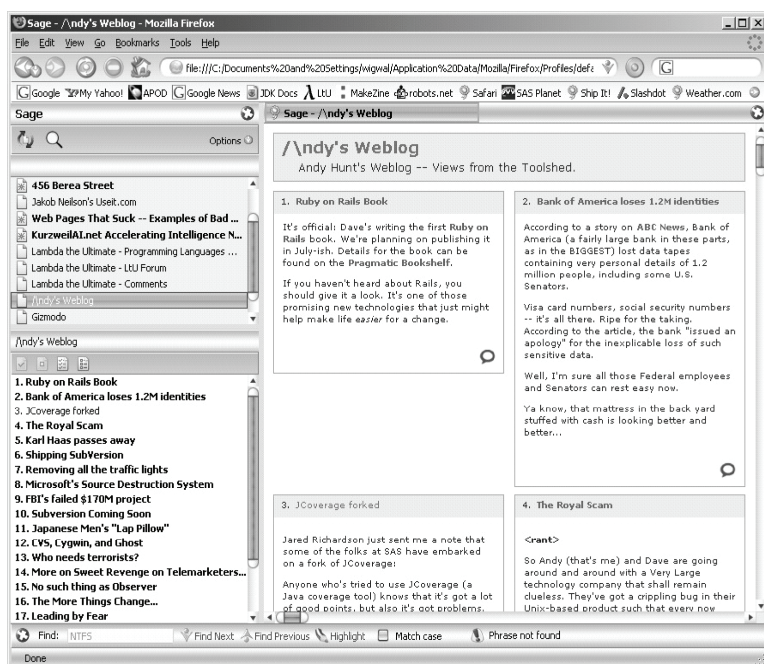


图 3-3　一个 RSS 新闻阅读器

绝对不要忽略你设置的优先级。在处理较低优先级的任务之前一定要先完成所有高优先级的任务，除非确实有充分的（而且得到大家认可的）理由暂缓某个任务。

时间估计

任务清单总有一个关联的时间表。这个时间表并非一成不变，但应当包括估计时间，指出任务清单中的各项任务大致需要多长时间完成。然后，当你完成一个任务时，要记录实际所花的时间，并注意二者之差。

过一段时间后，你（以及团队中的每一个人）都会非常擅长估计一个给定任务需要花多长时间。经过几轮反复后，技术领导人应该能根据单个团队成员的任务清单创建一个大致的项目时间表，项目经理也应该能做这件事情。估计时间时，任何答案都不算错。确实有些估计结果可能比另外一些更为接近。不用担心刚开始估计的结果差距太大。就像肌肉一样，使用越多就越强壮[1]。

---

① 如果你的团队估时有困难，可以尝试对选择做出限制。例如，估计的时间必须是一天、一周、两周或四周。开始时不允许有其他选择。

活跃

要做到有效，任务清单必须是活的，不能一成不变。你的团队必须能够适应变化。技术领导人会随着项目进展调整特性的优先级；一些新的特性会出现，而有些特性会黯然退出。优先级会改变，这是件好事，但开始时可能让人有些困惑，习惯就好了，要记住你的公司是在一个不断变化的市场中争取竞争力，这要求你也要灵活。对于这个变化的环境，不是要对抗，而是要适应。

实际上，任务清单有变化通常意味着你的客户和干系人在关注这个项目，而且确实提出了想法（和有价值的反馈）。大多数客户都会等到项目完成后才查看你的工作，但为时已晚。通常最好尽早得到反馈（尽管看到任务清单频繁改变可能让人有些郁闷）。如果任务清单在一段时间内没有任何变化，那么它可能已经无法反映项目当前的优先级了。

可测量

为了保证有效，任务清单上的每一项都必须是可测量的。毕竟，如果你想从任务清单中去除某项任务，就必须能确定这项任务已经完成。

基于这个原则，要避免一些模糊不清的任务，如“提高性能”，而倾向于一些具体的任务，如“保证登录在 5 秒内完成”或者“在 10 秒内生成报告 X”。通过创建一个只有“是”和“否”两种状态的目标，你就能明确这个目标是否已完成。而像“提高性能”这样的开放性目标会在产品的整个生命期中一直存在，最终成为一个黑洞。

如果你现在的任务清单上有一些任务是不可测量的，那么要花些时间查看真正的需求是什么。这个任务是否源于一个需要更快报告或更快启动时间的请求？把这个任务分解为明确的只有两种状态的任务，然后让原先提出要求的人检查这些任务。这个审查能保证你确实在解决客户的需求。

特性分段迭代而不是时间分段迭代

时间分段（time-boxing）的问题在于，我们向客户交付的是产品特性而不是日历上的日期。

你的团队在使用任务清单时，如果所有任务都按其优先级排序，那么你的任务清单就是按特性分段（feature boxed），而不是按时间分

段。管理层会查看任务清单，划出下一个版本中需要的特性。然后你为这些特性增加时间估计，并计算发布日期。你的产品和行业可能会规定要多长时间规划内部测试和 beta 程序，不过你可以具体地规划你的开发代码冻结（code freeze）。

销售团队决定必须加入某个给定特性时，这个特性在任务清单中的优先级可以改变，而且可以移入交付特性。不过，交付日期必须调整，要加入与这个特性关联的时间。

采用这种工作方式，可以为你的销售团队和管理团队提供一个清楚而明确的途径来了解特定特性与时间之间的权衡。他们不会再试图凭空决定交付日期和特性选择。不再毫无根据地强调需要更多时间的特性，他们会根据特定的时间限制权衡两个特定特性。

我们认为，采用这种方法可以在产品准备就绪时提供产品，而不是让你的公司随意规定一个发布日期（你很可能无法在这个最后期限前按时交付）。我们这个行业正是因贻误最后期限而闻名，从我们编写软件的方式来看这并不奇怪。不要试图将任意的特性集推入一个任意的最后期限，公司应当让开发团队指出他们能够做什么！如果开发人员无法达到销售人员期望的标准，那么现在就发现并调整计划显然更好一些，而不是错过了最后期限后才发现问题。如果开发团队可以很早达到要求，能早点知道不是更好吗？这样公司就可以向这个版本增加新的特性，或者把它更快地交付给客户。

第一次采用这种方式发布产品时，管理层会很紧张；第二次，他们就会放松下来；第三次，管理层就会完全相信他们的软件团队可以交付他们承诺的产品！

如果一项任务无法转换为可测量的目标，就把它设置为最低的优先级，先处理更高优先级的任务。如果这项任务的出发点是好的，将它完全删除可能是个错误，把它分解为可测量的任务就可以了。



小乔爱问……

什么是代码冻结?

代码冻结（code freeze）就是代码基停止改变。在开发周期中，代码如水般流动，会不断变化。不过，代码冻结之后，改变会停止。代码冻结后只能做重大的 bug 修正，增加特性或修正不太重要的 bug 都是不允许的。

“代码冻结”经常会降级为某种“代码半冻结”（code slush），此时版本中可能渗入不周全的变更。

针对性

现在你可能注意到了，我们既谈到了团队任务清单，也谈到了个人任务清单。这两类任务清单都非常重要，必须针对适当的对象。团队的任务清单要大得多，包含整个项目的所有重要工作。个人任务清单包含的任务较少（有时只是项目中的一项任务），但是一旦完成，就要从团队的任务清单中复制一项任务，把它放进个人任务清单中。

尽管很简单，但任务清单在很多方面都是一个很强大的工具。它能保证你有条理，不脱离正轨，还能保证管理层随时了解和参与确定你的工作方向。创建任务清单并确定优先级会让你全面考虑自己的工作，规划接下来的步骤。优秀的台球手会告诉你他已经规划好接下来的 8 次击球，好的开发人员也应该如此！

任务清单就像这样

下面是个人任务清单的一个例子，这里已经按优先级排序。

(1) 增加一个新报表，显示每天生产的部件。
(2) 增加一个新报表，显示每个员工生产的部件。
(3) 检查 bug #12345（查看 5 个月的报表时每月生产部件数显示为 0）。
(4) 在我的新工作站上安装开发工具。
(5) 查看非常棒的新报表……可能会补充到下一个版本的报表系统中。

注意最重要的任务要放在最上面。在这里，新特性比 bug 修正更重要，不

过也并非总是如此。另外，计算机升级和研究项目放在任务清单的最下面。这些任务只是用来填充停工间隙（也许你在等其他人的工作？），或者用于转移注意力，稍事休息。任务清单中有这些填充性质的任务很重要，这样就不会忘记低优先级的任务了。

如何起步

(1) 写出你全天要完成的每一项任务（最后这将成为你的“已完成”清单）。

(2) 把你的当日任务列表整理为一个正式版本的任务清单。

(3) 请技术领导人帮助你确定工作的优先级，并增加大致的时间估计。

(4) 开始处理任务清单中最高优先级的任务——不要违规！如果某个突发事件要求提高一个低优先级任务的优先级，一定要记录下来。

(5) 把所有新工作增加到任务清单。

(6) 完成任务时将这些任务移到已完成清单中（这样可以更容易地报告生存状态和“查找问题”）。

创建任务清单要求你对工作进行整理，并确定优先级。就像记日记一样，日记可以帮助你回顾和了解你做了些什么，而任务清单可以帮助你整理当前的工作，只不过采用的是一种高层次的轻量级方式。

每天早上要查看任务清单。只要有新工作出现就要进行更新……特别是最紧要的突发事件。不然如果有人问你上一周到底做了什么，你可能记不起来了。

做到这些说明你做得对

- ❑ 你的每一项当前任务都在任务清单上。
- ❑ 任务清单能准确地描述你的当前任务列表。
- ❑ 技术领导人或客户帮助你确定了任务清单的优先级。
- ❑ 任务清单可以公开获得（以电子方式或者任何其他方式）。
- ❑ 你使用任务清单来确定下一步要做什么。
- ❑ 你能很快更新（和发布）任务清单。

警告信号

- ❑ 因为你“太忙”，所以没有把任务增加到任务清单中。
- ❑ 更新任务清单比完成任务花费的时间还要多。

- 团队成员需要好几周的时间才能完成个人任务清单上的单个任务（提示：这些任务可能过于繁重）。
- 任务清单不到一周就更新一次。
- 任务清单上的优先级与“实际的”优先级不一致。
- 任务清单非常保密，团队以外的任何人都无法看到。
- 除了团队的任务清单，还有一些可以公开获得的不同版本。

11 技术领导人


小乔爱问……

如果我的团队不使用任务清单怎么办?

即使你的整个团队不使用任务清单，你也可以自己使用。把它写在你的白板上，或者一个标准笔记簿上，也可以写在你的 PDA 的 Wiki 上，或者写在你自己的网页上。请你的技术领导人帮助确定这些任务的优先级，并根据他们的反馈来确定工作的顺序。完成某项任务后就把这个任务划掉，每周重新整理任务清单。你个人以这种方式使用任务清单与整个团队都使用任务清单一样有效。最后优秀的技术领导人就会注意到你使用了任务清单，并且开始在整个团队中使用。你会惊讶地发现，好习惯会如此迅速地传播开来。这可能需要一定时间，不过人们总会注意到它的成效。

技术领导人对你的软件项目既要监督还要承担技术方面的责任。有了技术领导人，就可以把经理解放出来处理行政事务，而把技术方面的问题交给更胜任的人。经理可能身兼技术领导人的角色，不过这没有必要，很多情况下也不是一个好主意。如果你的经理缺乏必要的专业技术能力，或者你的团队在完成多个项目，最好有一个单独的技术领导人。

为什么需要技术领导人

你有没有过这样的工作经历？你的经理对你使用的技术一无所知，他们总设置一些不切实际的最后期限，根本不理解为什么拖延，对时间表斤斤计较。之所以会这样，就是因为他们不了解你在做什么，或者不知道技术如何使用。如果一个人从来没有做过开发人员，让他理解你的工作会非常困难。销售部门的人只知道如何使用你的产品，这可能没有什么问题，不过技术领导人就必须准确地了解你的产品是如何工作的。负责规划特性和制订开发计划的人必须很清楚代码做什么才有效。你需要一个技术水平高超的“联络员”，可以向不懂技术的经理解释产品和技术。你需要这样一个人在开发团队和管理层之间建立一个接口。所以，你需要一个技术领导人。

通常技术领导人就是一个上升到领导角色的团队成员。他们有开发背景，所以很清楚团队面对的技术问题。他们不会设定离谱的最后期限，因为他们知

道需要做多少工作才能满足一个“简单”的特性请求。技术领导人可以消除（或者至少能尽量减少）不懂技术的管理层的干预。

不懂技术的经理是一个极端，还有另一个极端是，可能有一个经理成天缩在办公室里写代码，根本不与管理层或客户打交道。这个经理是一个陈腐的开发人员，总是躲在办公室里，对与技术无关的东西毫不理会。你的团队构建的产品没有人想要，优先级也取决于经理的心血来潮，而不是客户的需要。一次又一次，你的团队构建的产品都被束之高阁。管理层不知道你的团队在做些什么，所以他们认为你们什么也没有做，这样一来，你的奖金、提薪还有升职都没有希望了。你的团队需要一个辩护人，能够花时间建立一些报表，展示给管理层、客户以及另外一些相关人员（他们对项目的感受很重要）。同样地，还是需要一个技术领导人。

技术领导人在这两个领域都要涉足。他们必须与开发团队合作，了解开发团队；另外还要与管理层、客户和其他技术领导人会面来交流你的团队在做些什么。技术领导人的位置很特别，他们花时间与产品的客户见面，并了解他们的需求，来确定你的产品应当做什么。

技术领导人需要完成以下工作：

- 确保团队的工作优先级与客户的需要一致；
- 确保将团队的工作适当地展示给管理层；
- 将团队与不懂技术的管理层隔离；
- 为不懂技术的干系人解释技术问题；
- 让开发团队了解非技术问题。



小乔爱问……

什么是干系人？

干系人（stakeholder）也称为利益相关人，就是对你的产品有所投入的人。设计一个产品时，要识别出产品的用户是谁，这很重要。你的客户可能是干系人，但是如果没还没有识别出干系人，可以由销售部（或者市场部）人员代替。在一些小公司，投资者往往就是你的干系人。有时在较大的公司里，干系人可能是另外一些部门，他们要在你的产品基础之上构建他们的产品。找出这些关键人物很重要，这样你才能构建他们真正需要的产品。

那么技术领导人必须怎样做才能完成这么一大堆任务呢？

技术领导人的职责

技术领导人有很多重要的职责领域，不同公司和团队组织中可能有所不同。以下是技术领导人最起码的职责：

- 为团队成员设定方向；
- 管理项目的特性列表；
- 为项目的特性确定优先级；
- 隔离你的团队，使他们不受外部干扰[①]。

下面进一步分析上述各项职责。

1．为团队成员设定方向

技术领导人就是团队的导师，要设定方向和优先级。技术领导人要与每个团队成员合作来创建和维护个人任务清单（见实践10）。技术领导人要了解团队成员的进度、遇到的问题以及估计的完成日期，并利用这些来建立项目健康度的全局视图，同时跟踪进展。这会成为一个联络点，可以提供一种快捷的方式，允许任何人了解准确的项目状态更新情况。

2．管理项目的特性列表

技术领导人相当于项目特性列表的主要保管人。所有特性请求都要经过技术领导人的筛选而不是直接统统交给开发人员。技术领导人会管理所有特性变更。

技术领导人处在一个特殊的位置上，他们不仅要了解每个特性的技术差别，还要清楚项目干系人的愿望和想法。技术领导人要根据需要增减项目特性，并指导团队的工作。

技术领导人和干系人首先需要创建一个特性列表来建立工作范围（在这方面，任务列表是一个不错的方法！）。然后，技术领导人与整个团队会面，估计每个特性需要的时间。有时这意味着要把一个大特性分解为多个列表项。最后技术领导人会向单个开发人员分配相应的实现任务，并建立大致的项目时间表。（注意，估计时间会随着项目的进展有所修改，有关内容稍后再讨论。）

如果有人不断地想在项目中增加额外的特性，这时让技术领导人管理项目

① 不过在这方面有矫枉过正的风险：过度隔离也会屏蔽一些很有价值的反馈。

特性列表确实很有帮助。技术领导人相当于一个缓冲区，会筛选请求，使每个请求得到一个合理的优先级。

技术领导人在为团队做掩护方面可能很有用。例如，大多数公司都有一些过于“有创意”的人物，这些人对产品总是有一些绝妙的想法，但是他们的想法往往会让你的团队分散精力。这种人总是在你最忙的时候前来拜访。

我们合作过的人当中，有一位主管 Ernest 尤其有创意，再没有比他更夸张的了。为了减少他的影响，我们让他把所有想法都告诉我们（而不是团队成员），我们会把他提出的新特性想法增加到团队的任务清单中（写在一个白板上）。增加这个特性之后，我们会把它与第一优先级的特性（必要特性）进行比较，我们都认为它不属于优先级最高的特性。然后我们再将这个特性与第二优先级的特性比较，依此类推。这类新特性最终总会排到第五优先级，通常只是可选的无用特性。

最后 Ernest 终于理解了我们的系统，开始把他的特性加到白板的最下面，放在第五优先级的特性下面。他已经将这些特性增加到任务列表中，所以我们不会忘记。实际上有些确实会增加到产品中，但是它们总会出现在下一版本中，而不是当前版本。采用这种方式对付这些过于有创意的人物，既可以把他们的想法输入系统，同时团队又不会受到这些新想法的干扰，也不会为增加无用特性而浪费时间。

3．为每个特性指定优先级

要为你的新列表设置一个顺序。如果没有一个既定的顺序，每个人可能都只会选择有意思的特性，而忽视那些真正必要的特性。要为各个特性安排正确的优先级，这几乎与在任务列表中增加适当的特性同样重要！幸运的是，这个问题可以交给你的技术领导人来完成。技术领导人必须与项目干系人合作来设置特性优先级。

干系人会对特性优先级产生很大影响，他们总是对项目及其成功抱有极大的兴趣。不过，干系人往往没有做出正确决策所需的技术背景。他们通常并不知道哪些在技术上是可行的。

技术领导人要与干系人合作来设定特性优先级。技术领导人可以做到，因为他了解团队的能力（这种了解会根据现实世界的反馈不断调整，最后趋于正确），还知道项目的技术细节。

在这个过程中，技术领导人会了解到干系人出于哪些非技术的理由请求某个特性。技术领导人要和干系人合作，通过协商共同设置特性优先级。这种协作对于没有技术经验的干系人或没有客户经验的技术领导人非常有用。

在实际中，一旦达到相互理解和信任，技术领导人就没有必要为每一个细节与干系人会面了。

优先级

优先级与数字相关联，第 1 优先级就是最高优先级，第 5 优先级则是最低优先级。当然，你可以根据自己的需要调整这些数字。我们目前就使用 1 到 5，不过在有些情况下，我们也用过 1 到 10。重要的是顺序，而不是数字范围。

- 第 1 优先级：必要
  这些是必须包含的特性，否则就不能交付产品。
- 第 2 优先级：非常重要
  即使不完成这些特性也可以交付产品，但是你可能不会这么做。
- 第 3 优先级：可有可无
  如果有时间，你可以完成这些特性，不过绝对不要因为这些工作延误交付日期。
- 第 4 优先级：精雕细琢
  这些特性可以为产品增加一种真正完成的感觉。
- 第 5 优先级：无用
  如果你有时间增加“无用”特性，说明你的进度超前而且没有超出预算。

4．隔离团队免受外部干扰

你正在做一个复杂的项目，一早上都“状态不错”，取得了不错的进展。就在这时销售部的一个人跑进来，问了一个关于下一版本的问题，把你的思路完全打乱了。光是看到这种情形就让你很恼火，是不是？不只是你，所有人都一样：如果不被打断，工作就会更有成效。实际上，研究人员指出，一个工作

日中多达40%的时间都可能因为中断而浪费[①]。这就像每天工作不到5个小时就下班回家！科学家们给这种现象起了个名字，叫做认知超负荷（cognitive overload）[②]。了解到这一点，技术领导人就必须尽最大努力保证团队的工作不受干扰。对此，一种很好的做法就是把技术领导人作为开发人员的联络点。一定要让技术领导人来缓冲这些干扰，不论它来自IT人员还是干系人。

列表项来自哪里

你使用什么为任务清单收集特性？可以使用需求文档吗？需求级用例？用户故事（或用户素材）？3×5卡片？大型政府合同？这些并不重要。重要的是能得到可以放在任务清单中的特性，而且为它们指定出优先级。任务清单不会取代你喜欢的需求收集方法。实际上，它会从各个来源收集信息，并采用一种简洁而且可以理解的格式表示出来。

重申一句，很清楚，任务清单不是要取代你收集需求的方法，而是取代（或增强）向团队和同事展示需求的方法。

技术领导人看上去像什么

技术领导人要把时间分摊到开发任务和管理任务上，而不能只停留在一个领域。技术领导人可能有几天甚至几个星期都泡在某一个领域里，但是大多数时间会分摊给这两类工作，二者兼顾。

技术领导人就是项目的守门人，他要保证团队成员不脱离正轨，要保证特性列表是合理的。技术领导人还要保证管理层了解团队的进展，确保客户的观点得到体现。技术领导人看上去就像一个全能选手。

自然，成也萧何，败也萧何，技术领导人可以成就一个项目，也可能毁掉一个项目（或者其中的一部分）。这可不是一件容易的工作，它要求技术领导人具有专业技术能力、沟通能力，还要有同时开展多项工作的能力。

每个项目都应该有一个好的技术领导人，而且每个开发人员都应该渴望成为一个好的技术领导人，起码要有一次这样的经历。即使你从来没有担任过这

---

① 研究人员已经确认，如果让一个人频繁转换环境，就会对他们的生产效率产生严重影响。这个消耗可达20%~40%。http://www.umich.edu/~bcalab/multitasking.html。

② David Levy博士研究了多任务并行会从哪些方面影响我们的健康和生产效率，参见http://seattletimes.nwsource.com/ pacificnw/2004/1128/cover.html。

个角色，掌握这些技能也会让你在开发团队乃至整个公司里变得举足轻重。大多数一流的技术精英在自己的职业生涯中都起码做过一次技术领导人。

> **技巧 15**
>
> 要有一个技术领导人

如何起步

如果你渴望成为一个技术领导人，那么首先要证明你已经做好准备，可以应对额外责任。仔细检查这个职位的需求，努力达到要求。要尽量主动完成技术领导人的更多职责。不要干等着这个职位落到你头上，而要展示出你在努力谋求这个职位，而且确实可以处理得很好。

对你个人的工作使用一个任务清单（见实践10），另外还要为团队维护一个任务清单。一方面监视工作的进展情况，同时还要关注即将进行的项目。

对团队的进展做出评价。找出弱点，寻找适当的实践或概念来解决问题，这些工作将为你提供一个新的视角。例如，如果团队在代码编译方面遇到麻烦，你就应当在自己的桌面计算机上构建一个 CI 系统（见实践 4）来帮助解决这个问题。

如果你没有马上被提升为技术领导人，也不要灰心丧气。要继续学习，努力提高自己，争取下一次得到任命。并不是每一个人都具有技术领导人必备的特质，不过朝着这个方向努力会让你对整个项目有一个更全面的认识，从而成为最有成效的团队成员。通过思考如何成为一个技术领导人，你会成为一个更棒的开发人员。

如果你刚刚成为一个技术领导人，那么首先要创建一个粗略的路线图。画出你的团队目前所处的位置，并标出你希望他们前进的方向。你要解决哪些问题？另外建议完成哪些工作？

为所有已知的问题建立一个清单。然后对团队做个调查，看看他们是否知道其他问题。如果你认为已经得到一个完备的清单，就要决定哪些任务可以解决，而哪些无法处理。

每日例会是一个跟踪团队工作而且不会让他们感到压抑的好办法（见实践12）。

做到这些说明你做得对

作为团队的技术领导人，你应该能顺利地回答以下问题：

- 你知道团队的每一个成员都在做什么吗？
- 你能不能在 5 分钟内生成一个关于项目状态的总结？
- 产品接下来要实现的 5 到 10 个特性是什么？
- 你能不能很容易地列出产品中优先级最高的缺陷？
- 你为团队成员解决的最近问题是什么？
- 如果一个团队成员需要解决一个重要问题，他会来向你求助吗？

警告信号

下面给出一些警告信号，如果存在这些问题，说明技术领导人的工作效率很低，或者过于专制。

- 缺乏对每一个团队成员工作方向的全局认识。
- 他一来，工作就要停下来。
- 团队工作出色，但只有他得到好评。
- 不能解决问题，或者更糟糕地，反而会带来问题。
- 不能准确地预测工作时间表。
- 不清楚团队成员的技术能力，也不知道团队成员希望了解什么。
- 对团队中的冲突视而不见。

12 每天都要协调和沟通

大多数人都不会喜欢太多的会。如果你的团队每周开一次会，每次会议可能至少有一个小时的时间要用来讨论通知、上周工作和下周计划。每日例会则完全不同。这只是简短的团队会面，可以鼓励交流和沟通，而且不会对进度带来很大影响。

每个团队成员会简要地告诉大家他在做什么，遇到了哪些问题。对此有一个很好的经验：每个人花的时间不要超过 1~2 分钟。要记住，这个会议会召集整个团队，所以要注意烧钱率，要力求简短，切中要点。



小乔爱问……

什么是烧钱率？

烧钱率（burn rate）是一个术语，描述运营公司要花费多少资金，包括工资、租金、电力、福利等等。这是你要“烧”的金额，与你的工作是否完成无关。不管召开多大规模的会议，一定要先退一步，对这个会议每小时的花费做一个大致的计算。知道这个数额会让会议开得更简短。

有一个很好的经验，假设每个开发人员每小时耗费 100 美元（要记住，我们不仅要考虑直接工资，还要考虑间接开销）。这说明，如果你的团队有 10 个人，那么烧钱率大约就是每小时 1 000 美元，一天就是 8 000 美元，一周则是 40 000 美元。下一次开会时，如果推迟了 10 分钟才开始，或者整整用了 30 分钟听 Fred 大谈他的度假见闻，那么计算一下成本吧。如果你的产品拖延了三个月，这个开发时间又会给你的公司带来多少花销？要记住，这还没有考虑浪费的时间内贻误机会可能带来的开销。

为什么需要每日例会

要解决不能很好地沟通这个问题，最容易的方法就是更频繁地与团队交流。这会帮助你了解哪里存在问题，从而有机会做出修正。

大多数人都会朝着团队的目标努力。遗憾的是，人们往往会理解有误或者失去方向，所以你必须采取措施加以调整。

要与整个团队更频繁会面，让每一个人告诉大家他在做些什么。其目标是频繁进行航向修正：如果你在路上开车，你不会设定一个方向后就任由它行驶。你会指出想去哪里，然后做大量微小的修正，根据需要向左或向右转动方向盘。如果太长时间未能做出修正，就可能冲出路面，车毁人亡。软件项目也是一样，如果太长时间没有做小的修正，项目就会失败。

技巧 16

通过每日例会频繁进行航向修正

每日例会有什么作用

从召开每日例会的第一天起，你就能体会到它带来的诸多好处。

又脱轨了

有些类型的开发人员总是偏离航向。最常见的就是那些没有经验的开发人员和“风滚草型”开发人员。新加入的团队成员了解的情况很少，会花大量时间来解决早已经解决的问题。风滚草型开发人员要年长一些，也更有经验，不过他们总是频繁地偏离航向。每日例会可以帮助这两种人不脱离正轨。

重复创造

没有经验的开发人员会解决一些不需要解决的问题。充满活力的年轻开发人员很少遇到他们不能解决的问题，可惜的是他们解决的问题往往早就被其他人解决了。很遗憾，团队成员通常不会向这些刚毕业的新人做足够的交待，你要把这一类信息告诉他们。这些新员工不是从同事的错误中学习经验，而是继续重复创造，做无用功，导致计算机科学的发展不断反复，几乎没有进展。

如果开发人员实现了一些编程语言中早已经提供了的数据结构，你就会意识到这个问题。这些开发人员可能创建了漂亮的 GUI 部件，可惜另外一个开发人员上个月也写过同样的部件。最后对于你能想到的每个工具，工作室里的每个人都编写了自己的版本。如果大家一起讨论过，这些工具就可以只写一次，其他人都能共享。但这个工作室的开发人员都太忙于写代码了，根本没有时间来交流。结果，工作室里每个开发人员编写的 GUI 部件外观都稍有不同，还分别有自己的一组字符串处理例程。

如果你的团队每天会面，最新加入的开发人员 Lee 告诉大家他开始编写一

组新的 GUI 部件。Ellen 就会告诉 Lee 去年她和 Mike 写过一些部件。那些部件可能并不完全满足 Lee 的要求，不过可以作为他的工作的起点。Lee 可以在其他团队成员建立的基础上工作。利用每日例会，你就建立了一个论坛，可以做这种讨论，而不是最后等到 Lee 费尽周折快要做完时，才偶尔在午饭时或者在休息室里向某个人提到这些 GUI 部件。

风滚草型开发人员

我们都与风滚草型开发人员共事过。这些开发人员没有方向，成天“漂来漂去”。他们不辞劳苦地编写随机的代码，并努力加以“改进”，比如清理方法签名、优化算法、调整括号的格式。风滚草型开发人员缺乏纪律，往往不完成你要求的任务，带来的危害总是多过好处。

我们认识的一个风滚草型开发人员喜欢清理方法签名和删除未用的参数。例如，经他之手，

```
doSomething(String foo, Integer bar)
```

会变成

```
doSomething(String foo)
```

遗憾的是，像大多数风滚草型开发人员一样，这个开发人员从来不修改使用 `doSomething` 的代码，所以他的修改往往会导致构建失败。

风滚草型开发人员喜欢改进算法来提高速度，但是他们的修改会导致代码生成错误的结果。他们会花数小时调整代码的格式，来适合他们的口味。风滚草型开发人员有一大堆自己的习惯，只是因为那是他们认为“正确的事情”。

风滚草型开发人员很容易控制。首先，召开每日团队例会。其次，关注风滚草型开发人员的每日报告。如果他们开始偏离正轨，好的技术领导人就要及时发现问题，免得他们走得太远。最后，技术领导人应当确保这些风滚草型开发人员每天都有足够的工作量。这种策略不能完全杜绝他们的游离，不过可以尽量减少他们的空闲时间，并限制他们的破坏。

专业技能放大器

如果你的团队里有几个高级成员，每日例会可以充分利用他们的专业技能。每次他们分享一个解决方案时，整个团队都能听到。另一方面，每当一个低级

成员谈到某个问题时，这些高级成员也能知道并提供帮助——如果不知道问题，当然无法帮助解决。

例如，假设 Ted 参加了每日例会，说他在开发一个 XML 解析器，但处理国际字符时遇到了问题。Mike 恰好去年遇到过同样的问题，会告诉 Ted 如何解决。Ted 很快得到问题的解答，而不必自己花几个小时来追查这个问题。每个团队成员都可以利用整个团队的经验快速解决问题。

团队沟通

每日例会让每个人都说一说，但不会让任何人为难。每天把大家召集在一起讨论和分享想法，这会让你的团队惊人地团结。有多少团队能够做到每个成员每天至少与其他所有成员讨论一次？每日例会可以把这些分散的开发人员凝聚成一个真正的团队。

从每日例会得到好处最多的人是那些过于羞涩、成天躲在办公室里的开发人员。曾与我们共事的一个人（暂且把他叫做 Rick）就是这种性格的典型代表。Rick 上班后，径直走进他的办公室，一直工作到午饭时间，午饭也是独自在他的办公室里吃，然后下班时悄悄离开。大多数时间他根本不与任何人谈话。

一旦开始每日例会，所有团队成员——甚至包括 Rick——都会相互认识。会前、会中和会后都充满欢声笑语。开发人员相互帮助来解决问题。大家建立了浓厚的友情。最后我们听说，Rick 已经不再羞怯沉默，开始与团队中许多人交往，邀请大家吃午饭，甚至偶尔会找别人“聊聊”。说实话，原先我们根本想象不到他能这么积极地与人交往。

我们看到，每日例会在很多人身上都产生了这种效果。尽管这些人起点不同，但都取得了积极的结果。

全局视角

通过开会讨论其他团队成员在做什么，可以避免“功能性近视”。如果团队成员有这种“近视”，他们会认为自己的工作是唯一的，自然也是最重要的。这就导致各种幻想自大症，让开发人员对自身的地位过于高估。通常，只要让这些人加入一个团体，对各个项目和工作方向进行讨论，就可以将他治愈，而不需要更多其他治疗。

除了治疗“功能性近视”，知道其他团队成员在做些什么，你的整个团队也可以从中受益。如果 Jeff 和 Mutt 知道 Jen 开始重构一个组件，而他们的代码

正在使用这个组件，他们就可以计划处理其他代码，这样 Jen 的工作就不会影响到他们。如果 Jen 的重构与 Jeff 原来做过的工作很类似，Jeff 还可以提供帮助。可以看到，知道各个团队成员在做什么，分享有关的信息，会带来各种有益的副产品。

其他方法

除了团队会议，还有很多其他候选方法。下面就来看几种。

经理或技术领导人可以不时四处走动，顺便走访团队成员。这样会为每个团队成员提供与技术领导人一对一交流的时间，这是一件好事。不过，这无法提供整个团队的交流。而且对于技术领导人来说，这需要耗费大量时间，他们每周要花一到两天的时间四处走访。最后还有一点要注意，技术领导人不再作为屏障来保护团队不被打断，他自己反而成为导致中断的人。

另一个候选方法是让每个人单独工作一两个月，然后再把大家召集起来。尽管这种做法得到广泛使用，但我们并不推荐。这会导致巨大的工作量，而且开发周期最后还需要大量加班，却得不到太多可用的产品。如果你的团队在这样一个“真空”里工作，沟通不良的问题就会被放大。如果你偏离了正轨，可能在浪费了数周甚至数月的时间后才能意识到。我们就曾在以这种方式运作的工作室工作过，多次看到开发人员几个月的辛苦工作付之东流。

每周例会也非常流行。每周例会面临这样一个问题：分享多大的信息量才能保证每周例会真正有效。团队成员在一周内已经做了太多工作，不可能再出现有意义的信息交换。实际上，每周例会将成为经理共享信息或发布通知的一个论坛。优秀的经理可以利用这个论坛，让团队关注某个特定问题，但是如果他们不知道有些问题已经存在，就无法关注。每周例会比季度例会要好一些，但是仍然不及每日例会能为团队带来的好处。

每日例会就像这样……

- 早上 8:55：团队成员开始纷纷走进会议室。这个例会是所有人一天日程中必不可少的一部分，每天都在同一时间同一房间召开。会前时间用来讨论一些“重大问题”，比如哪家咖啡馆的咖啡师最棒，山地自行车技术的最新发展，或者超酷的自制软件 MythTV 项目等等。

- 早上 9:00：技术领导人 Maurice 宣布会议开始。有些时候这意味着必须打断某几个团队成员热烈的交谈，他们可能正在兴高采烈地讨论他们的最新山地自行车。一个团队成员开始做工作状态报告，这一次先从 Chris 开始。

  Chris从昨天开始为产品增加对一个新数据库的支持。他在一些SQL语法上遇到一个小问题，不过已经隔离并修正了它。此时另一个团队成员插话进来，他提到另外几个同样不可移植的SQL命令的有关技巧。现在Chris知道了团队中有人之前移植过SQL，以后如果他遇到问题就可以找他们帮忙。
- 早上 9:03：Tiffany 坐在 Chris 旁边，所以接下来轮到她来讲。Tiffany 正在编写一个客户应用原型，这可能成为团队产品的新前端应用。Tiffany 花了一天时间为这个应用编写主屏幕。她预计今天能完成 GUI 布局，午饭后应该可以开始增加 GUI 的底层逻辑代码。
- 早上 9:04：Mike 正在升级一个 PC 机，他花了一天时间重新安装软件，并重新设置这个机器的首选项。另外还完成了一个简短报告。
- 早上 9:05：Fred 正在调试客户报告的团队产品中的一个问题。表面看来，团队的服务器产品一到星期一早上就定期崩溃，不过在一周中的其他几天都运行得很好。这时 Jake 插进来，开始讲他完成的另一个产品的故事，那个产品曾经存在类似的问题。那时，数据库服务器一过周末就会重启，他的应用在星期一早上尝试访问数据库服务器时产品就会崩溃。Jake 继续为大家讲述整个故事。几分钟后，Maurice 打断了他，故事暂告一段落。Jake 和 Fred 商量好会后再深入地谈一谈，一起查找这个问题。
- 早上 9:10：Maurice 向大家公布了几个通知，包括星期五的一个聚餐，还介绍了刚刚得到的一个新客户。通知要放在会议最后发布，因为总有人迟到，如果在会议一开始公布，他们就会错过。
- 早上 9:15：散会，在 15 分钟的简短会议后，大家都回去开始工作。

如何起步

如果你以前从来没有开过每日例会，那可真够呛！下面是启动每日例会的几点想法。

- 一定要让每一个人都知道模式（你希望哪些问题得到回答）。

- 每个人都必须回答问题。没有人能跳过，无一例外。
- 开始时，在时间限制上可以宽松一点。最开始会有大量新信息交换，所以必须允许沟通自由顺畅地进行。
- 会议要在每天相同时间相同地点召开。让每日例会成为一种习惯，而不是勉强坚持的例行公事。
- 把每日例会时讨论的话题发布在网页或 plog[①]上。
- 挑选一个人开始会议，然后顺时针（或逆时针）轮流发言。一个人讲完后随机选择另一个团队成员发言比较容易让大家心理紧张。

目标集中

如果每个人都不离题，每日例会就会成为一个很有价值的工具。会议必须保证很具体，这很重要。不要说你“已经完成 70%”。相反，应该说你今天增加了登录屏幕，虽然还没有提供具体功能，但是明天就会增加相应功能。比如，如果有人说登录屏幕有问题，可以稍稍打断，让他们更详细地描述一下这个问题（例如，“它无法与认证管理器通信”）。对于一个新项目，可以由技术领导人主持会议，不过最后应该让团队中的每一个人轮流主持。利用这些每日例会在内部培养领导人。

当一个话题开始变得过于深入，或者会议开始变成解决问题的讨论，领导人就要迅速转换话题，让有关的团队成员在主会之后私下进一步交流。如果 Jim 和 Sue 要对一个只有他们才有的问题做半个小时的头脑风暴讨论，没有必要搭上整个团队一起陪听。他们可以在找出解决方案之后为大家提供一个简短的总结。

很多著名的开发方法坚持让每个人都站着，以此缩短会议时间。这是有效的，不过如果参加会议的人经过培训，可以保证发言简短（或者领导人经过培训，能够让每一个人都不离题），就没有必要要求大家都站着。可以试试这两种办法，看看哪一种更适用于你的团队。

---

① plog 就是一个项目博客（project blog）。博客即 Web 日志。而 Web 日志就是一个在线日记，设计的目的是便于更新。通过使用 plog 来共享每日例会的信息，更容易让团队成员“掌握内幕”。这是一个集中的信息来源，可以让没有参加会议的团队成员、其他团队或经理了解有关信息。



小乔爱问……

我们的每日例会时间太长了。该怎么做呢?

刚开始举行每日例会时，往往会持续较长时间。每日例会要共享信息，而且有很多事情要做。你的目标是保证每个人讲一到两分钟，但是最初几天甚至几周都不太可能做到如此简洁。还要考虑到每次加入新的团队成员时时间会拖得更长。每日例会是一个让他们了解最新信息的绝好论坛！

如果经过几周之后你的每日例会仍然要持续一个小时，这就说明存在问题，需要解决。你很可能让团队成员过于详细地解释问题和解决方案。不要让他们过多地纠结修正工作的细节，要尽量只提供总结。例如，不必对问题、调试周期和最终解决方案提供一个底层的详细分析，只需要这样说，“我们在缓存方面遇到一个问题，数据修改后无法更新。现在这个问题已经解决，而且已经签入。”这就是我们需要知道的全部。

还可以要求团队成员写出他们打算共享的信息。这会帮助他们在会前整理想法，避免长达 5 分钟“杂乱无章的报告”。

还可能存在另一个问题：参加会议的人太多。我们发现每日例会只能扩大到大约 15 人。如果人太多，要想办法把每日例会尽可能划分为更小的组。让同一领域的团队成员参加相同的会议。一定要保证至少有一个或两个人有重叠，同时参加不同领域的会议，以便传递相关的信息。

做到这些说明你做得对

如果已经有每日例会，那太好了！下面的原则可以确保你方向正确。

- ❑ 这些例会有用吗？如果团队中没有人能从中了解到任何信息，说明报告可能过于简练了。如果某个领域需要更多详细信息，可以针对这些主题分出一个更小的组开一个小会。不过，两分钟原则只是一个指导原则，而不是硬性的法则。你可能发现 30 秒就足够，有时也可能需要 3 分钟。
- ❑ 例会是每天都在相同的时间和地点召开，还是总是不断变动？在同一时间同一地点召开每日例会更容易让人记住。会议偶尔可以有变化，但是要避免频繁变动。

- 如果你停止召开这些例会，人们会不会抱怨？他们应该会！团队应当依靠每日例会来“了解最新动态”。如果例会可以取消，说明它们根本没有价值。团队应当把每日例会作为一个意义重大的资源。

警告信号

每日例会是一个很好的工具。但是像任何工具一样，如果操作不当，也可能产生危害。下面的警告信号表明每日例会已经偏离正轨。

- 每个团队成员需要十分钟或者更多时间。
- 某个团队成员总是占用太多时间，几乎是其他成员时间的总和。
- 人们以一种不友善的方式相互责问。团队成员之间开开玩笑是很好的（也是值得鼓励的），但如果每日例会变成相互攻击的场所，就不会有任何成效。要排除那些肆意攻击的人。
- 会议总是很晚才开始（或结束）。
- 会议变得空洞无物，开发人员只是宣称“我完成了 90%”，或者“我在做关于 Frozbot 的工作”。
- 团队成员在漫无目的地聊天，忘记要报告他们做了些什么。你要私下里要求这些团队成员把他们做的工作写下来，这样在开会时他们就能保证目标集中，报告简洁。他们还可以建立自己的任务清单从而更有条理。

13 审查所有代码

频繁做小规模的代码审查可以保证代码清晰、简单而且整洁。可以避免传统的让人不快的代码审查，这些审查往往涉及数十个开发人员，而且需要好几天的时间准备［也称为“极其糟糕和可怕的代码审查”（The Mighty Awful and Dreaded Code Review）。为了不影响你阅读，后面把这种审查称为 MAD 审查］。我们发现，只要遵循以下原则，代码审查就可以非常轻松：

- 只审查少量代码；
- 最多有一两个审查人员参与；
- 经常审查，一天数次。

你的目标是逐渐养成更频繁审查代码的习惯，同时不会出现结对编程（pair programming）可能带来的文化冲击（或者明显增加开销）。很多环境无法适应结对编程这种程度的交互，单单是为了让呼吸带着薄荷气息所带来的成本就可能毁掉一个原本蒸蒸日上的公司！所以，可以不采用结对编程，而尽量更经常地审查代码，而且每次只审查较少的代码。

结对编程

结对编程就是让两个团队成员在同一台计算机前工作。一个键入代码，另一个后退一步，更关注全局。一个人处理代码的细节和语言语法，另一个人则确定一个特定算法是否合适，能否用来解决某个问题。第二个人要发现问题，如编码错误、拼写错误和变量名不当。这两个开发人员还要不时交换角色。

有些人很喜欢这个实践，不过有些人完全不接受这种做法。我们发现，如果使用得当（而且用于适当的人），这确实是一个很有用的实践。关于这个“奇妙”的实践，可以访问网站 http://www.pairprogramming.com/（网站名确实很贴切）更深入地了解。

如果一周都没有做一次代码审查，就意味着你留出了大把时间让一些严重的问题有机会肆意潜入你的代码。如果这段时间内你一直在处理一个棘手的问题，那么可能需要一个外部视角。这个人甚至可以不用懂太多，你只需向他解释一遍问题，这个行为本身通常就足以解决问题［《程序员修炼之道》把这个人

叫做橡皮鸭（rubber ducking）[1]。］如果等几天才做一个代码审查（即使只是一个临时检查），这很可能会变成一个耗时很长、很痛苦的经历……一个 MAD 审查！

要避免 MAD 审查，需要把你的工作划分成尽可能小的部分，独立地审查各个部分，并提交到源代码存储库。如果某个领域存在问题，这就很容易隔离。

程序员可能很容易陷入某个特定任务的细节当中，以至于贻误了明显的整体改进。如果你停下来向另一个人解释你的工作方向和代码，就要中断原来的工作进程，通常会得到很有价值的全新观点。有时我们可能太忙于在森林里开辟道路，却没有意识到实际上这并不是我们要的那个森林，我们一直在错误的方向上披荆斩棘！

将代码分段还有一个好处，如果代码划分成更小的部分，审查人员就更容易理解代码。在一个快节奏的开发工作室里，代码可能要一天审查多次。不过，在这方面有一个好经验：千万不要工作两天以上而不做一次代码审查。要把代码审查当成呼吸。当然，你确实可以屏气几分钟，但是谁愿意这么做？

理想情况下，每增加一个特性（或者修正一个 bug）都要有一个审查。如果增加了 7 个特性另外修正了 14 个 bug 之后才对代码进行审查，就很容易陷入可怕的 MAD 审查（更何况还有之前旷日持久、毫无重点的辛苦努力）。

如果你正在重写产品中一个复杂的部分，无法将这个任务划分成更小的部分，那么可以找一个审查人员，让他频繁地实时对代码做临时审查。

如果你知道有别人在查看你的代码，并要求你对代码负责，你就会写出更好的代码。这不是开发人员独有的问题，而是人的天性。代码审查可以确保至少有另外一个人检查你的工作。你很清楚：这种问责制之下，你不能在代码中走捷径。

大量研究表明，代码审查在检测代码缺陷（bug）时非常有效。实际上，这是查找 bug 的头号技术。再没有比这更好的技术了。如果你没有坚持不懈地做代码审查，最后发现的问题可能会让你大惊失色。

我们确实见过类似 mrHashy（Mister Hashy）的变量名，用来表示一个散列表。不过，经过一次代码审查后，开发人员就开始使用更合理的变量名，以避免再次受到同事们的嘲笑。同伴的压力可能让人很痛苦，但确实很有效。

---

① 之所以这么叫是因为另一个人并不需要对这个谈话做任何贡献，只需要在适当的时候点点头就可以了。如果你找不到一个合适的人，甚至用一个橡皮鸭也是可以的。

“橡皮鸭”（前面解释过）是一种查找和解决问题的有效方法。向某个人描述你的代码时，你会突然意识到原先忘记的事情，发现有些逻辑行不通，或者可能与系统中另外某个方面存在冲突。希望你在每次签入代码时都和橡皮鸭“谈一谈”。

除了做“橡皮鸭”，其他开发人员还能发现你的代码中的 bug。由另外一双眼睛检查你的代码，往往能捕获到你自己从来不曾注意到的问题。你会有一个完全不同的视角。在开发工作室里查找 bug 往往比实际应用中现场查找 bug 开销低。这个小小的投入会带来巨大的回报。

代码审查对于促进团队成员之间的知识共享也很有意义。经过审查中的协作，审查人员至少对你的代码做什么有了概念上的认识，而你希望能有更详细的了解。这在指导方面有很大好处，而且对代码维护也很有帮助。

代码审查为经验丰富的开发人员提供了一个绝好的机会，可以借此向缺乏经验的程序员传授编码风格和设计技术方面的经验。除了明确一些小的技术细节（如括号放在哪里），代码审查使有经验的老手有机会向新手们建议为什么应采用另一个数据结构，或者指出出现了一个模式。通常，在这些会话中审查人员会发现重复的代码或功能，可以移至公共基类或工具类中。这样在代码签入到源代码管理系统之前可以得到重构。

模式

模式是指记录和命名正式项目中常见问题（及其解决方案）的实践。学习模式的原因有很多。其中一个原因是可以为开发人员提供通用词汇表。开发人员一起工作后，会形成一组通用术语，使他们能很快而且无歧义地沟通。模式可以快速启动这个过程，即使是你刚遇到的人，也能与他准确无误地顺利交流(假设他们也熟悉同样的模式)。

学习模式的另一个理由是，这可以帮助你解决以前从未见过的问题。通过阅读和讨论不同模式，你将学会如何解决很多常见问题。关键并不是你是否会遇到大多数模式，而在于当你遇到这些模式时能不能明确识别出来[①]。你知道如何干净利落地解决这个模式表示的问

---

① Edsger W. Dijkstra 的 *The Humble Programmer* 是一篇研究计算机科学（包括模式）发展的经典文章（实际上，这是他获得图灵奖时的演讲稿。——译者注），即使今天仍然很有适用性。要知道这篇文章写于 1972 年！（参见 http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF)。

题吗？还是跌跌撞撞做过多次代码迭代后才找到一个可接受的解决方案？

《设计模式：可复用面向对象软件的基础》（作者 Eric Gamma、Richard Helms、Ralph Johnson 和 John Vlissides，也称为“四人帮”）可以作为学习模式的一个很好的起点。

审查还有利于对小的编码细节以及全局概念进行交叉培训。除了“编码规范”，你还会学习“如何编写规范的代码”。

下面的指导原则可以帮助你完成代码审查。

代码审查必须至少包括另外一个开发人员。现实中，几乎总是只有一个开发人员，除非你创建的代码很有意思或者很巧妙，其他团队成员很想了解。这种情况下就可以放手让更多开发人员加入审查。不过，不要走极端（最多不要超过 3 个到 4 个开发人员），太多开发人员会使审查陷入困境。

如果没有经过审查，就不要公开代码。完成审查之前，不要把代码变更增加到构建产品的源代码中。代码签入时应该包括一些注释，其中列出审查人员的名字。这样一来，如果对代码变更的原因存在疑问，而你不在跟前，就能有第二个人可以做出解释（至少能做出基本解释）。

重构

Martin Fowler 对重构给出了描述，我们无法对这个经典描述再做改进：

“重构是一种纪律性的技术，可以调整既有代码的结构，修改它的内部结构而不会改变其外部行为。其核心是一系列小的保持行为不变的转换。每个转换（称为一个重构）做的很少，但是一系列转换就会产生显著的结构调整。由于每个重构很小，所以不太可能出错。系统在各个小的重构之后依然能够正常工作，从而降低系统在重构期间被严重破坏的可能性。”[①]

不要把这个代码审查原则当作不签入代码的借口。如果你的公司有一个源

① 摘自 http://www.refactoring.com/。

代码系统，其中只存放生产代码，那么可以把你的代码放在一个私有区，直到准备就绪可以发布。这个私有区可以是一个单独的代码存储库，也可以是另外安装的一个源代码管理系统。尽可能不要让代码只存放在你的机器中，这样当你的机器崩溃时（因为机器很容易崩溃），你的代码不至于随之消失。

审查人员有权力拒绝他们认为不可接受的代码。当你审查某个人的代码时，如果没有正确地注释，或算法效率低下，不论出于什么原因，都不要顾虑，一定要要求他做出修改（不过不要过于吹毛求疵——要记住，条条大路通罗马，达到同样的结果往往有很多可以接受的方法）。作为一个审查人员，你的任务是改善代码，不是不经审查就一味批准。正如 Eric S. Raymond 所说："很多双眼睛会让虫子无处遁形。"

如果你对代码的解释无法让审查人员理解，那么代码必须简化。作为一个审查人员，如果你不理解或不认可，就不要表示同意。毕竟，你的名字会与这个代码关联在一起。正如《程序员修炼之道》指出的："要在你的作品上签名。"要确保它值得你写下自己的签名。

任何代码变更都不能破坏现有的自动化测试。（你确实在做测试，对不对？参见实践 7。）如果你还没有运行过测试，就不要浪费同事的时间要求他们做代码审查。如果你需要更新现有的测试，可以在审查之前先完成这些测试的修改（作为你的编码工作的一部分）。增加的所有新测试也应当成为审查的一部分。作为一个审查人员，如果你认为还需要更多的测试，一定要拒绝代码变更。

"首先不要造成伤害"[①]并不只是一个代码审查规则，因为这是一个普遍适用的一般原则。代码变更绝对不允许破坏产品。当然，如果有一个很好的测试套件，这个规则就会存在争议，但是这并不能作为破坏现有功能的借口。不应该破坏一个现有的 API，而应当增加第二个 API，在其中包括你需要的额外参数（或其他任何内容）。

例如，如果必须改变一个现有的函数调用，就应该建立一个进度表，对现有例程的删除做出规划。不要背着你的客户（同事或者公司的其他团队）悄无声息地将这个例程删除，首先要对是否保留原来的例程做一个理智的决定。对删除工作做出规划也很重要——如果一个例程已经存在多年，就不要贸然废弃它（你要有自知之明！）。

① 出自希波克拉底誓言。

审查人员需要轮换，但是不要太过严苛。有时让同一个审查人员连续审查也是可以的，不过要避免所谓的“伙伴系统”，比如说你总是审查 Kevin 的代码，反过来 Kevin 总是审查你的代码。另外，不要整个团队都找同一个指定的（常常是超负荷工作的）审查人员。这两种情况都会影响你预想的“异花传粉”[①]效果。

代码审查不必正式。不需要安排一个会议，只需就近问一个团队成员，问他是否有时间做一个审查。有时甚至在代码编辑期间就可以进行审查。有时你可能会打印出代码改动部分让审查人员带走。格式或地点并不重要——只要能够进行审查。

引入代码审查过程时，你可能需要任命几个高级团队成员作为主管审查人员，其中一个高级团队成员最开始必须参与每一次审查。这个角色不必保持太长时间，不会超过几个月。一旦你的团队成员学会了基本方法，整个团队都能分担这个责任。正如圣经中所说：“铁磨铁，磨出刃来。朋友相感，也是如此。”[②] 重点是让团队成员协作，从而共同提高。要尽快让你的团队成员参与到这个“磨刃”过程中来。

我们工作过的一个工作室就是一个很好的例子，充分体现了如何使用代码审查来充分发挥高级人员的作用。我们有 3 个高级开发人员，另外还有 5 个成员肯定不能算是高级开发人员——他们并不完全是新手，不过有时他们对如何修正一个问题一点想法都没有。为了保护产品，也为了让这些低级开发人员上一个台阶，所有代码审查都包含一个高级团队成员。这样更资深的团队成员可以指导并传授经验，另外在低级开发人员向产品中引入问题之前就能发现问题。采用这种方式，高级团队成员还可以了解低级开发人员的误区和遇到的问题。

这些审查对这个团队帮助很大。我们经常发现重复的代码，这时就会立即将其取出，移入到工具类中。审查人员会捕获并删除对指定工作没有任何帮助的代码（这也称为“否弃重构”），并断然拒绝未注释的代码。团队的努力使得一个变化悄然发生了，尽管察觉不到但非常重要。

低级团队成员都开始养成好习惯，一次完成一个代码审查。无需明确要求，他们就会在审查之前开始整理代码，增加有意义的变量名和注释，诸如此类。

---

① 思想的“异花传粉”是指向着新的可能性和新的做事方式拓展。——译者注

② 《旧・箴》27:17, NIV。

冗长、繁琐的例程变得简短、易于管理。

更棒的是，大家记住了代码审查中学到的教训。过了大约三个月，我们改变了代码审查策略，任何团队成员都可以完成审查。

如果审查中有一两个开发人员总是有遗漏，就应当使用代码变更通知反复检查他们的工作（见实践 14）。监视代码变更通知可以提供一个非侵入性的简便方法，来监视工作室里的任何成员，而不用呆在办公室里站在他们背后盯着。

有时你可能全神贯注地考虑某个问题，根本无法分心停下来参与一个代码审查。参与审查后要想把心思再转回到这个问题上来需要耗费大量时间。（还记得关于中断的讨论吗？见实践11）。如果你正专注于某个问题，此时有人进来要求做代码审查（或者其他事情），你要告诉他们你现在还“拔不出来”，让他们以后再来。另一方面，如果你正在找一个审查人员，而有人说他正在专心做另一件事，那么你可以先走开，等他手头的工作结束后再找他，或者干脆另找其他人。

虚拟代码审查

经过一段时间，你会了解特定的审查人员会查找哪些方面的内容。例如，Jared 曾经写过一段相当复杂的代码，而且成功运行，这让他很满意。然后他完成了一个“虚拟审查”，想找出他的两个最高级的同事可能针对什么问题进行审查。Jared 考虑了他们分别会有什么建议，根据他预想的这些建议实施修改之后，才让这两个审查人员实际审查代码。他们非常满意！这三个人一同审查过太多的代码，所以 Jared 能从他们的角度分析。他了解那两个开发人员（比他多好几年的工作经验）最看重什么，并能利用这一点来改进自己的工作。这正是完成代码审查的初衷：你不仅在构建好产品，同时也在培养优秀的开发人员。

与软件开发有关的很多工作都是脑力劳动——一个问题会一直在我们脑海里盘旋，直到最终解决。如果你正处在需要全神贯注的情况下，让别人稍后再来并不是侮辱他。有些工作室里这是自然而然的，但在有些工作室看来则完全无法接受。让别人 30 分钟后或者午饭后再来通常都是可以的。

技巧 17

可以说“以后再来”

管理层必须要求进行代码审查。如果没有管理层认可，你的工作室里没有人会名正言顺地主动参加审查。换句话说，如果没有要求某个人来帮你，他们可能不会为你花时间，特别是在快到最后期限时间很紧的情况下。

即使你的工作室没有一个强制性的代码审查政策，你仍然可以请团队成员对你的代码进行审查。整个团队不会因此得到好处，不过你自己的代码会改善。过一段时间后，审查代码的人也会了解到代码审查的好处。

如果想让一个人帮你做审查，但他暂时没有时间，那么不要等他太久。可以在工作室转一转，看看有没有别人刚好能抽出身来。如果必要的话，还可以走得更远一些，但是一定要找到一个可以做审查的人。如果你找到的人之前从来没有为你做过代码审查，可以利用这个绝好的机会让他们了解你做了些什么。

这些简短的代码审查可以促进知识技能的传授，而不存在正式培训的开销。要与不同的开发人员共同完成代码审查，这样一来，你就会得到不同开发人员的经验和专业技能，让你大有收获。各个审查人员可能会提出不同的方法来解决同一个问题。有的方法好一些，有的可能欠佳，不过都各有千秋。

技巧 18

经常审查所有代码

我们的目标是学习如何创造性地思考，同时改善你的产品。要学习从不同角度查找自己的问题。经常做这些简短的代码审查，过一段时间后就会变成一种“第二本能”，就像微波炉一样，你可能奇怪原先没有微波炉的时候是怎么过的。在审查中关于算法分析或资源约束进行的讨论，将成为你学到（并牢牢记住）的经验教训，因为你实际应用了这个经验。

你不是在学习一本学术理论的书，也不是要考取证书，而是坐在工作台前。这里聚集着很多工匠，有些是师傅，有些是学徒，大家相互学习，你可以用他们的技巧充实自己，直至某一天你自己也成为师傅。

如何起步

代码审查是非常棒的工具！一旦养成习惯，你就会奇怪没有代码审查怎么

可能写出高超的代码。可以使用以下的技巧作为起步。

- 要让每一个人都了解你计划做哪种类型的代码审查。要频繁地审查，而且每次只审查较小的代码块。不要等到几周后，已经积累了数百甚至上千行代码变更之后才做审查。不要让你的团队陷入 MAD 审查！
- 在前几周或几个月里，要让一位高级团队成员参与每一个代码审查。这是共享知识的好办法，而且可以使审查有一个可靠的基础。
- 确保代码审查是轻量级的。宁可审查太少代码，也不要太多。完成两个重叠的审查比完成一个较大的审查更好。
- 要引入一个代码变更通知系统（见实践 14“发送代码变更通知”）。这是对代码审查的一个很好的补充，而且有助于提醒忘记申请审查的团队成员。
- 确保在要求所有团队成员参加之前得到管理层的认可。

做到这些说明你做得对

- 代码审查会自动通过吗？除非团队中的每个人都尽善尽美，毫无遗漏，否则不应该发生这种情况。
- 每个代码审查都会带来大幅度的重写吗？如果是，说明某个方面存在问题：可能是编码人员，或者是审查人员，也可能是技术领导人（即为编码人员和审查人员指出方向的人）。
- 经常做代码审查吗？如果审查的间隔时间以周为单位，说明等待的时间太久了。
- 你总是在轮换审查人员。
- 你从代码审查中学到什么了吗？如果没有，就要在代码审查期间开始问更多问题。

警告信号

- 代码审查不频繁。
- 大多数代码审查都很痛苦。
- 人们不愿意签入代码，因为他们不想做代码审查。
- 审查代码的团队成员无法解释代码能做什么或者为什么写这个代码。
- 低级团队成员只审查其他低级成员的代码。
- 类似地，高级团队成员只审查其他高级成员的代码。
- 某个团队成员成为每一个人想找的首选审查人员。

14 发送代码变更通知

编辑代码时，自动构建系统可以注意到变更，并重新构建项目（见实践 4）。下一步是发布这个信息，使团队的每一个成员都知道这些变化。

变更通知系统会把这个信息推送到整个工作室，而不只是你身边接触的同事。这种知识共享的效果可能相当惊人。

类似于 Alistair Cockburn 说的“信息辐射器”，你也在向外提供信息。团队其他成员可以使用这个信息，也可以不理会，总之信息已经给出，可供大家使用。实际上，这个实践并不是让你获取数据，而是把数据推送给你。

信息辐射器

Alistair Cockburn 如是说：

“信息辐射器会在一个任何路人都可以看到的地方显示信息。有了信息辐射器，路人不需要再问问题，信息会在他们经过时显示给他们。

“一个好的信息辐射器有两个特征至关重要。第一个是信息要随时间改变，这才值得人们花时间查看显示。另一个是查看显示所需的能量要很少。”

摘自 [Coc01]。

每次我们引入这个实践时，工作室里很大一部分人一开始总是极力反对。

大约一个月之后，原来意见最大的人总是回来道歉，告诉我们这个工具变得对他们多么有用。这个技术刚开始总是遭到最强烈的抗拒，但是稍过一段时间，所有人都会习惯于这些通知。很快这会成为一个必不可少的资源。

下面是一个典型的代码变更通知：

```
In the DB package, added createRecord() to TdDataFile

Index: com/tde/db/TdDataFile.java
===================================================================
RCS file: c:\apps\cvs\cvsroot/WeissDB/com/tde/db/TdDataFile.java,v
retrieving revision 1.3
diff -r1.3 TdDataFile.java
381a382,401
```

```
>  /**
>   * 返回表中存储的记录类型的一个新实例。
>   * @return TdRecord - 记录的一个新实例。
>   * @exception ClassNotFoundException - 若找不到存储的类名则抛出该异常。
>   * @exception InstantiationException - 若创建不了存储的类名则抛出该异常。
>   * @exception IllegalAccessException - 若表示存储类名的类范围不对，或不存在零
>   * 参数构造函数，则抛出该异常。
>   */
>  public TdRecord createRecord() throws ClassNotFoundException,
>            InstantiationException,
>            IllegalAccessException {
>      // Create a new instance of the record
>      return (TdRecord) (Class.forName(getRecordName()).newInstance());
>  }
>
>
```

实现这种系统有两种方法。首选的方法是将变更通过电子邮件自动发送给每一个团队成员。大多数自动构建系统都会为你发送变更（往往还会把变更发布到一个网页或 RSS 提要上）。实现这种系统的另一个方法是手动完成。每个团队成员需要隔离出他们的代码与老版本文件之间的差别，并把这些差别通过电子邮件发送给团队的每一个人，我们把这个过程称为“邮件发送差异”（mailing the diffs）。

不论采用哪种方法实现，每次代码签入到源代码管理系统时，都应当将变更通知发送给你的团队。通知邮件应当包括以下内容：

- 审查人员的名字；
- 代码变更或补充的目的（例如，你修正了哪个 bug，或者增加了哪个特性）；
- 新代码与老代码之间的差别（任何主流的源代码管理系统都会为你生成这个报告）。如果你完全重写了一个相当大的代码块，那么只列出差别就会毫无意义（因为二者差别过大），此时只需包括新代码。这一点同样适用于新文件。

意外的好处

很多年前，我们重构过一个相当复杂而且运行时间很长的算法，想让它运行得更快一些。完成重构后，我们照惯例把代码变更通知发送给团队。过了一个小时左右，一个团队成员来找我们，他几乎完全重写了整个算法。他的修改

比我们的代码要快一个数量级。

这个团队成员拥有高级算法分析方向的博士学位。所有人都知道他的背景，但是这几个月实在太忙碌了，没有人想过让他来审查变更。不过，他看到了这封邮件并读了注释块，这引起了他的注意。利用专业背景，他立即发现了对整个算法的一个“明显”改进，而别人从来没有想到过。这个故事就是一个典型的例子，说明代码通知可能会带来意外的好处。代码变更通知确实可以（而且通常会）取得出乎意料的好处。

让每个人都知道

代码通知是在团队成员之间培养责任感的一种简便易行的方法，有助于找出特立独行的开发人员，他们不愿做代码审查，或者增加的代码并不对应任务清单中的某个 bug 或特性。

你很快就会注意到最积极的编码人员和审查人员。如果有人一个星期都没有提交代码，技术领导人就可以去看看，确保他没有陷入困境或者脱离正轨。我们曾在一家公司根据请求把 CEO 加入到代码通知列表中，使他能够紧密监视“项目的一举一动”。对于大多数软件项目来说，让管理层了解情况很困难，而这就是一种可以帮助推送信息的简便方法。

在作品上签名

《程序员修炼之道》提醒我们要有一种为自己的工作而自豪的工作态度。不论是设计软件解决方案还是盖教堂，都应该这样工作，就好像你处理的每一件工作都要在明亮的灯光下经过同事和客户的仔细检查一样。如果你知道你打造的木板将作为教堂的前门，而不是一个小储藏间的背板，是不是会有不同的工作态度呢？

在你的作品上签名并不是指你有绝对的所有权，或者任何其他人都不能编辑这个代码。这只是表示这个代码由你完成，而且你将对你的作品提供支持。如果有人要问有关这个代码的问题，你是回答问题的第一人选，第二个将是审查代码的团队成员。如果有人发现代码存在一个问题，他们可以知道是谁完成了这个代码，谁可以修正问题。

> 要对你的作品很精通，这也包括要成为你完成的那部分代码的专家——这没有任何问题。医生还分各科，术业有专攻，大多数职业也是如此。也许你会找你的家庭医生做每年的体检，但是一旦发现问题，你一定会尽快去找一个专科医生。希望团队中的每一个开发人员都能了解每一段代码，在真实世界的项目中是不切实际的。在你的作品上签名，等于宣布了你很清楚而且可以处理这部分代码。强制开发人员玩“抢座位游戏”并不能确保每个人都成为所有代码的专家，其后果只能是任何人都无法成为任何代码的专家。有些情况下，让所有人都保持同一水平可能会有好处，不过通常我们更希望身边有专家，不论是代码专家还是医疗专家。

如果你愿意，也可以不理会这些通知

你也可以不理会这些通知邮件。不过，最终你可能发现自己会不时地瞄一眼，想从中得到一点信息。也许你不知道，实际上你已经非常依赖这些通知，并且每天都会使用它们。

我们的编辑 Andy Hunt 审查这个手稿时，他把这一节标出来，认为有疑问。(在出版界这就表示“删除!”。——编者按）非常偶然，还没等我们找时间与他讨论这一节的意义，他在访问一个客户时看到这个技术得到了非常成功的使用。在他看到技术得到使用并取得了实际效益之后，这一节终于得以保留。毕竟，如果一个技术有效，它就是实用的。

如何起步

引入代码变更通知有很多方法。手动系统和自动系统我们都用过。采用手动系统时，要手动地键入邮件内容（手工地贴入代码差异部分），并把它们发送给团队成员。采用自动系统时，会有一个程序监视 SCM 并生成通知，发送邮件。

这两种方法都是可行的，但自动系统往往更可取。不过，在你的环境中建立一个自动系统可能很困难，如果是这样，也可以使用手动方法。

在通知到来之前一定要让你的团队有所了解。

做到这些说明你做得对

通知必须定期发送，而且值得信任。

代码差异部分不要太庞大（比如说，多达 5M）！

警告信号

这里唯一需要注意的问题是可信任度。通知必须是可信赖的。如果团队成员不相信代码变更后会发出邮件，他们当然不会依赖这些通知。这个问题采用自动系统比手动系统更容易修正，不过不论使用哪种方法，都必须考虑到这个问题。

15 大汇总

不同于那些更复杂、功能更全面的方法，上述这些方法是你确实可以做到的。从现在就开始实施，你会立刻看到好处（至少有一些）。当然随时间推移你还会看到更多好处。在工作方法上投入得越多，你看到的好处就越多。世上并没有无所不能的“银弹”，但这里展示的技术确实可以帮助你避免很多常见的灾难，同时尽量不干扰你的工作。实现这些技术会让你看上去像是一个编程高手，而实际上你只是充分利用了其他人的经验。就像老话所说，你会“站在巨人的肩上”。

下面对这一章讨论的技术做一个总结。可以把它复制下来，贴到墙上，并按章行事。很快你就会奇怪，使用这些技术之前你怎么会那么做事。

- 任务清单
  - 可以公开获得
  - 已经指定优先级
  - 有一个估计时间表
  - 活跃
- 技术领导人
  - 管理项目的特性列表
  - 跟踪开发人员当前的任务和状态
  - 帮助指定各个特性的优先级
  - 隔离团队免受外部干扰
- 每日例会
  - 保证简短
  - 力求具体
  - 列出问题，但不要解决
- 代码审查
  - 只审查少量代码
  - 一两个审查人员
  - 经常审查
  - 不经审查不能发布代码

- 代码变更通知
  - 用电子邮件发送并发布通知
  - 列出审查人员的名字
  - 列出代码变更或增补的目的
  - 包含代码差异部分，如果篇幅允许还可以包含文件本身

第 4 章 曳光弹开发

夜晚用机枪射击时，很难看到子弹飞向哪里。即使可以清楚地看到目标，在黑暗中命中也不是一项容易的任务。对机枪手来说很幸运的是，有人发明了曳光弹。曳光弹经常会与常规子弹合为一体。曳光弹包含少量发光物质，发射时会燃烧，在空中留下一条长长的弧形轨迹。

那么曳光弹对于软件开发有什么意义呢？

发射曳光弹时，你可以准确地看到它的去向。这就使你能够在真实条件下实时地调整瞄准目标，使子弹准确地命中。

曳光弹开发

曳光弹开发（Tracer Bullet Development，TBD）对软件项目也有同样的作用：采用这种方法，项目一开始你就能看到走向，它可以帮助你从项目早期开始就连续瞄准目标。TBD是我们见过的开发软件最有效的一种方法——这种方法非常易于使用，而且功能极其强大。实际上，正因为它如此重要，我们将用整个一章来讨论这个实践。

开发过程（不论是TBD还是其他过程，比如RUP或XP[①]）相当于一根线，可以把工作的各个方面都串在一起。它把实践方法、工具和技术等联合成一个聚合的整体。

过程就是一组步骤，这些步骤连接在一起就能采用一种可重复的方式构建产品。利用过程，可以多次用同样的方式、同样的工具构建产品，这样一来，开发工作不再是一个碰运气的赌博，而变得可以重复并值得信赖。销售和市场团队向你提出产品想法后，你的团队每次都能交付产品。也许不能按销售部门

① 关于不同方法的更多详细信息，参见附录F。

理想的时间表来交付，但是公司完全可以相信：你肯定能在承诺的时间范围内交付你承诺的产品。

流行的过程方法存在的问题

如今有很多流行的过程方法，不过大多数都过于复杂，令人畏惧。

有一类过程方法就像提供了一个虚拟盛宴，你必须从数百种实践中做出选择，还要分配数十个角色。遗憾的是，各种选择往往太过复杂，仅仅为了知道该对哪些实践进行评价就必须聘用专业顾问。这些实践的实现和用法都同样困难——大多数实践人员在管理这些系统上花的时间甚至与他们写代码的时间一样多。

另一类过程方法表面上很灵活，告诉你要“定制”自己的过程，然后却“出尔反尔”，坚持让你使用他们的核心实践（而且坦率地讲，有时这些实践很奇怪）。你可以使用你喜欢的任何实践，但前提是从他们的建议实践中选择。这些过程方法的追随者努力把他们的“外来文化”强加给你的工作室。把文化强加于人总会失败，但这些追随者反而会责怪你。因为他们认为这是“正确”的做法，所以罪责肯定在你身上。

不论哪一类，引入这些新实践往往会让整个工作室动荡不安，以至于管理层再也不敢碰任何新的过程方法，永远不会。

最后，相关的人（你的客户和最终用户）并不关心你使用什么过程。他们只想知道你是否交付了一个可靠的产品，是否还可以再次做到。

你的过程只要回答两个真正的问题：

- 它对你有效吗？
- 它是可持续的吗？

定义你的过程

毫无例外，我们见过的每一个没有明确定义过程的工作室都无法按时交付有期望特性集的产品。不仅因为这种工作室不清楚自己在做什么，更与缺少过程方法有很大关系。不要让这种情况发生在你身上：要花些时间搞清楚你在做什么以及为什么这么做。

作为实际经验，一个过程方法如果不能容纳其他最佳实践，几乎可以肯定这是一个不好的过程方法。如果一个过程或方法声称对所有项目中的每一个问

题都是独一无二的解决方案，一定要当心。这种魔法般的“百宝丹”只是现代的“万金油”。应当采用一种鼓励定期重新评价而且能够加入适用于你项目的实践方法。一定要保证你的过程是一个灵活的过程——要看一个新的、更好的实践是否适用，不要害怕改变或调整你的过程。如果你有一个新想法，可以尝试几个星期。如果确实能很好地工作，那太好了！让它永久地补充到你的过程中。否则，要进行修改或者干脆去除。你要通过这种试验来找出哪些方法对你的工作室最适用。好的过程中没有“神牛”[1]。只要适用就可以保留，任何不适合的东西都必须删除或修改。

每个项目都不同，而且每个团队也是独一无二的，所以你是唯一有资格评判哪些适用而哪些不适用的人。

例如，考虑四周迭代（four-week iteration）的想法，也称为 sprint[2]。一个工作室使用这个实践，计划每四周有一组新的可交付产品。我们看到管理层虔诚地坚持这个标准，尽管开发人员和测试人员都一致抗议，表示在他们特定的环境下根本没有足够的时间。这个工作室应当考虑试试五周迭代，或者三周迭代。而且谁说过迭代必须是紧接着的呢？如果觉得紧张，可以在迭代之间稍做停顿，做一些测试、bug 修正，或者为下一轮进行头脑风暴。

真正的目标并不是四周迭代，而是提交卓越的软件。如果一个很著名的实践在你的工作室里并不适用，就要做出调整。没有什么不能改进。保证敏捷就意味着必须调整那些无法较好地适应特定需要的做法。

存储数据

通常你会遇到这样一种情况：你需要可再生的输入或输出数据，特别是对于测试和曳光弹开发。我们把预定义的数据称为存储数据（canned data）。存储数据通常用于测试，因为它可以为测试提供可再生的输入。在曳光弹开发中使用存储数据，可以使组件返回“合法的”数据，尽管组件还没有提供完备的功能，但看起来也能工作。

技巧 19

目标是软件，而不是遵从过程

① 对印度人来说，牛是一种神圣的动物，不可冒犯。——译者注

② sprint 指一段时间，在这段时间内特定工作必须完成，并准备接受审查。——译者注

带着以上需要注意的问题，我们来更深入地了解曳光弹开发。可以以此作为你工作室的工作基础，或者原封不动地照搬使用。重要的是，要知道过程是什么，还要有一个经过深思熟虑的过程。要保证这个过程足够轻，以避免“优柔寡断”或者“基础设施超负荷”，但是同时也要足够详细，以保证工作不脱离正轨。TBD 力求在这二者之间达到很好的平衡。

TBD 如何工作

TBD 并不试图改变你的工作方式，它只是“包裹”已有的工作方式，并力求做到不侵犯你的其他实践，与它们无缝地共存。这也是最小的可用过程之一，非常容易使用。

使用 TBD 时需要创建一个端到端的工作系统，但是系统组件都是中空的对象。你要为系统的所有重要组件编写代码，不过这些对象不做任何工作。例如，登录例程可能只接受一个用户名（如 Fred），并生成一个合法的登录。数据库访问层会返回数据，但是返回的实际上是存储数据，而不是来自数据库的数据。巧合的是，在项目的早期需要完成每个对象的内部组成，也就是说它们在“等待”完成（to be done），所以 TBD 这个缩写真是很贴切。

模拟对象

这种做法是用哑对象（dummy object）替换系统中还不实用（可供测试）的对象。例如，如果你在一家大型电信公司工作，可能并不是全天候都能访问一个价值五百万美元的交换机。不必等到轮到你访问交换机时再来测试你的代码，可以使用一个模拟对象（mock object）作为交换机。对于大型数据库、昂贵的硬件等通常就会采用这种做法。想了解更多信息请访问 http://www.mockobjects.com。

曳光弹开发是这样做的。

明确项目的主要部分，把产品分为相关功能块。例如，可能有名为客户（client）、Web 服务器（Web server）和数据库层（database layer）等的块。

定义这些块需要交换哪些信息，并用方法签名记录这些信息。这些层间的交互称为接口。不要奢望前几次就能建立完美的接口。

把各个块交给不同的开发人员、开发人员团队或者你的不同大脑分区（如

果你单独工作）。

只需编写让一切看起来能正常工作的代码。可以把这想成是模拟对象的一个完整应用。每一层看起来在验证用户或获取数据（或者在完成你的应用需要做的任何工作），不过实际上每一层都是“桩”，只是在返回存储数据。

有了这个“瘦”的骨骼框架，接下来就可以开始在各个块中填入实际的逻辑代码。

最后，也是最重要的，要记住在整个项目过程中接口会改变和发展。你的第一发子弹往往会漏掉目标，所以要灵活，适当调整你的瞄准点。其他团队要求新接口或者要修改现有的接口时，你就应当做出修改。毕竟，这是软件。

接口

我们使用接口这个词时，并不是指C++、C#或Java接口。这里是指程序层用来传递信息的例程。接口是程序域之间的 API。这是两个不同领域的交互或接口点。

定义系统对象

第一步是明确你的应用可以划分为哪些层（对象）。客户、服务器和数据库就是这种对象的很好的例子。当心不要定义底层对象。在客户服务器应用与基于 API 的数据库中，对“底层”的定义显然会大不相同。

要确保你定义的每个系统对象都可以独立存在。如果创建了一个与系统其他部分之间有清晰界限的对象，这就可以是一个系统对象。稍后我们将看一个例子。

要保证这些对象尽可能大，团队才能独立地工作更长时间。如果此时定义系统中的每一个对象，就会浪费大量时间；这个阶段为这么多底层对象（如 Person 或 Address 等对象）创建和维护接口会有些过分。

系统对象必须足够大，足以让个人或团队单独花一段时间处理。还必须能够在各个系统对象之间创建清晰的界限。日志管理器就是系统对象的一个很好的例子。日志管理器有一个明确定义的 API，能够由很多其他系统对象重用，可以在后台重写而客户无需知道这些修改。

数据库管理器也是系统对象的一个很好的例子。为数据访问定义一个 API 时，要封装其余的系统对象，而不暴露存储和获取数据的细节。也可以对数据库管理器完成某种重写。也许你会切换数据库，或者只是为了提高性能而做出调整，不过如果没有一个清晰的 API，就必须调整其他系统对象。



小乔爱问……

曳光弹可以处理复杂的系统吗?

在这个例子中，我们展示了一个非常简单的架构，每层有一个系统对象［有时这称为“图腾柱”(totem pole) 架构］。另外，这个例子在不同层分别有一个对象是为了简单起见，并不是TBD的要求。

作为一个例子，假设你的项目是这样一个应用：请求一个服务器获取数据、进行分析，然后返回结果。在这个场景中，有 4 个主要的系统对象：

- 客户层
- Web 服务器层
- 数据处理层
- 数据库访问层

在这个阶段通过建立非常大的对象，可以将项目的主要部分解耦合。例如，我们的示例应用有一个 Web 服务器层、一个数据处理层和一个数据库访问层。通常这三个对象都“纠结”在一个名为“服务器”的对象中。如果只有一个服务器对象，我们分离到这三个层中的所有功能将通过一组服务器接口相互交织在一起。使用一个解耦合的架构时，不同的团队就能很容易地在不同层并行地工作，也能更容易地自动化完成验证各层功能的测试套件。

可以把服务器对象想成是锅，而把开发团队想成是厨师。当然，如果锅足够大，每个人确实可以用同一口锅。不过，如果每个人都有自己的锅，自然要容易得多。

不同的开发团队分别在不同层工作，每个团队可以认为自己的层位于一个不同的计算机上。这意味着各层之间的所有通信都通过网络完成，这会给你带来很多好处。

首先，没有人能够绕过明确定义的接口直接访问你的代码。(在重要的最后

期限前一天晚上经常会发生这种情况！）不同层在不同的机器上运行，这就提供了一个无法逾越的牢固屏障。

这种架构的第二个好处是可伸缩性。某一层需要更多资源时，可以把它移到一个更大的机器上。在传统的紧密耦合的架构中，除非做重大重写，否则不可能得到这种可伸缩性。多个（甚至是所有）层最后可能都在同一个机器上，但是这不再作为一个要求。如果某一层开始对性能有影响，就可以把它移到另一个机器上（稍后介绍有关内容）。

如果越过这么多细节使用这些大对象让你感觉不安，也不用担心。向系统对象增加功能时可以创建细粒度的对象。如果你想这么早就在系统中定义每一个对象，那么在证明整体设计能正常工作之前，可能会耗费整个团队大量的时间。在最好的情况下，团队成员会耗费时间来了解某些用来表示地址的对象（本来他们根本无需了解这些对象）。在最坏的情况下，这甚至会浪费整个团队的时间来学习最终将舍弃的对象。要尽可能保证底层代码"等待完成"（To Be Done，TBD）。

协作定义接口

接下来，处理相邻层的团队要会面，共同明确两层共享的接口。如果你知道客户应用需要登录，就能知道 Web 服务器层中必然存在一个登录调用。团队要协作并对方法名和签名达成一致，然后编写各个方法，但是只返回存储数据。例如，可能有一个哑登录例程，如果你的名字是 Fred，这个登录例程就会成功通过，而对于其他用户名都会失败。

在这些会议中，要开始定义各层相互之间如何通信。你会使用 CORBA 编写一个本地应用，还是编写一个 HTML 客户与一个 servlet 通信，或者编写一个 C#应用与 ASP 服务器通信？Web 服务器使用 CGI／Perl 还是使用 HTML／Ruby？你的团队要共同确定这些细节，使得这次会议（或者可能是多次会议）之后，每个相关的人都能知道并理解各层之间的接口。

可以看到，你要完成很多工作，而不只是在这些会议中记录方法签名。在层、服务器类型和通信细节的定义过程中，你也定义了你的架构。

最好的架构并不是由"架构师"在象牙塔里凭空定义的，它们是合作的产物。不要让某个专家来推动，并把一个完整的架构文档摆到你面前，而是要你的团队一同合作，充分利用并增加每一个人的经验。在这个过程中，如果有一个架构师帮助主持讨论当然有很多好处，但是一个没有最新产品经验的架构师

“在真空中”设计的产品绝对会是一场灾难。

设计技巧

下面的技巧可以让你的接口设计会议顺利进行。

- 总有一个人主持会议。这个人拥有发言权，任何人讲话之前必须经他“许可”。让某个人主持会议有助于避免会议变成一场争吵。
- 整个会议中应在白板上记录要点。由于信息写在白板上，每个人都能立即看到大家认可的方法签名。如果在纸上记录，肯定有人无法看到你写了什么。
- Andy Hunt 建议用 LEGO 或积木表示系统中的对象。如果能提供一些看得到、摸得着的实物，就能帮助更多低级成员理解系统以及不同对象之间的关系。有时由于你的工作缺乏实体性，所以很难可视化表示和理解系统组件。不论是在白板上画出对象，还是在桌子上移动积木，都可以为系统建立一个可视或可触摸的表示。
- 记录接口并发布。可以使用一个打印文档、Web 页面或者 Wiki，但是不论采用哪一种媒介，都必须保证信息公开。对象的接口绝对不能作为秘密。
- 保证会议不被中断。要尽量减少转移话题和回答问题的次数。

这种集体参与的架构设计有很多好处，如下所列。

- 定义产品的过程会成为团队成员的一个学习过程，特别是低级团队成员。
- 团队共同创建架构时，会得到对整个系统的全面了解。
- 团队会对他们设计的系统有很强的主人翁意识。如果只是交给你一个规范（由一个与世隔绝的僧人般的架构师所创建），你绝对不会有同样的感觉。

这里的目标是避免权利被剥夺，很多开发人员在被要求编写代码而不是考虑问题时都有这种被剥夺权利的感觉。如果把你的架构强加给你的团队，而且有些开发人员的正当权利被忽略，开发人员就会感觉自己像是一个机器中的齿轮。尽管一线编码人员非常了解技术，但是没有办法改善整个系统，这种感觉可能让人极其郁闷。让每个人都加入进来，就可以构建一个更好的系统，同时

还能培训你的团队成员。

技巧 20

集体参与建立架构

编写接口桩

这是项目中最容易的部分。要记住一切都应力求简单。接口的目标就是要足够“瘦”以便编译和使用。

如果服务器需要一个 Login 接口，那么你的代码应当只包含正确的参数和某种成功指示（如一个返回码），除此以外不应该再有其他内容。例如，登录调用可能如下所示：

```
Boolean Login (String userName, String password) {
    return True;
}
```

增加总线数

总线数是指只有当损失的开发人员达到这个数，项目才会失败。不论这些开发人员是辞职还是被公车撞了，总之如果失去这么多人，你的项目将无法正常进行。

如果你的团队中有一个“超级明星”，项目的所有详细信息都在他手里，那么总线数就是 1，这就会带来问题。如果团队中每个成员都可以替补其他任何人，那么只有当失去整个团队时项目才会出问题。理想情况下，总线数最好等于团队成员数，即使做不到这一点，至少也要努力增大这个数。如果失去一两个关键人物就会破坏你的产品，就应当采取措施增加这个数。

曳光弹开发会自动增加你的总线数。相邻层的团队成员一同定义他们的共享接口时，其实就在共享各层操作的有关知识。由于两个团队在共享这些知识，这使得团队成员可以更容易地从一层移动到另一层。至少，每个人都对相邻层做什么以及如何做有一个基本的了解。

如果你从未考虑过项目的总线数，现在就请花几分钟来调查你的团队。你的总线数是多少？可以采取哪些步骤来增大这个数？

注意这一节中的代码示例都是伪代码，没有采用某种特定的语言。在设计时使用伪代码是为了避免针对语言语法发生“圣战”（holy war）。这里我们强调的是接口，而不是具体实现。

可以增加例程 `AnalyzeDataSet`，它接收一个数据集（但会将其忽略），然后返回一个“已分析”的小数据集。当然，两个数据集都是存储数据。可能如下所示：

```
DataSet AnalyzeDataSet (DataSet thisDataSet) {
    DataSet thatDataSet = new DataSet();

    thatDataSet.add(2);
    thatDataSet.add(3);
    thatDataSet.add(5);
    return thatDataSet;
}
```

你并不关心用户是否登录，也不在意“已分析”的数据集是否正确。你要的只是一个客户可以调用和使用的例程，这就是这个阶段的目标。

插入增加功能的代码之前，一定要把所有接口走通一遍。要抵制诱惑，不要那么轻易就开始编写代码。一旦增加了一点功能，就会更难抵制诱惑不去增加下一个功能，以及再下一个功能。接下来的事情你就知道了，处理相邻层的团队成员无法编译他们的代码，因为你的接口尚未完成。

如果你认为非常有必要在某个领域记下一笔，以免忘记一个绝妙想法，那么可以添加注释。可能甚至还可以记录你认为将成为例程主要部分的内容，或者某个潜在问题的有关警告。这样既可保留信息，又不会深陷在具体代码中。

一旦完成这一步，你的接口（或API）就已经建立，各个团队会对他们在项目中的角色有一个清楚的认识。所有人现在都知道他们必须做哪方面的工作，而且已经有一个骨架可以填入具体代码。

实现层间通信

既然已经完成了桩，现在可以开始让它们与系统中的其他层通信了。

这听上去可能并不重要，但是你会惊讶地发现，很多看上去不重要的细节并没有按照人们预想的方式运作。一旦开始增加回调，使用不同的 CORBA 开发商，并尝试通过防火墙或类似设施在不同的子网使用 Java RMI，就完全不像“Hello, World!”那么简单了。

如果较早地捕获到这种不兼容性，你可以及早改变架构，而不是等到团队基于这些假设编写了 10 000 行代码之后才大动干戈。同时，如果经过 6 个月的开发之后，整个系统根本不能像你声称的那样工作，你的老板肯定会对你有意见，而尽早做出调整可以避免这一切不良后果。

你的客户代码现在可以访问服务器，并要求它验证登录凭据。给定前面所示的桩，你知道登录请求总会成功。在这里只要验证客户确实可以与服务器通信，而且服务器可以传达登录请求是否成功的信息。

既然客户在与服务器通信，服务器也需要访问其相邻层。在这个登录调用中，你的代码可以向数据库访问层请求与用户登录有关的信息。

```
Boolean Login(String userName, String password) {
    String realPassword = getPassword(userName);
    Boolean login =  False;
    if(realPassword.equals(password)) {
        login = True;
    }
    return login;
}
```

当然，数据库访问层不会具体查看任何人的信息。对于每一个密码请求它都会返回存储数据（例如 asdf）。（这也会在接口设计会议备忘录中留有记录。）

增加这个内容后，我们现在已经让客户与 Web 服务器通信，另外 Web 服务器会与数据库访问层通信。换句话说，你有了一个曳光弹，它由客户发射，经过主要的层后返回。你只是证明整个系统能够“通信”！每个团队在他们的桩中加入代码来访问其他层时，就是在证明所涉及的各个技术确实可以互操作。现在你已经到达了一个里程碑，要知道很多重大项目在编写了数千行毫无用处的代码之后才能做到这一步。

但是在把系统放入一个真实的生产环境之前，先不要太过自信。客户会不会在一个防火墙后面运行部分系统？如果是这样，那你也要这么做！

我们并不是说大楼里的每一个工作站都必须是生产环境的一个镜像。不过，至少应该有一个这样的工作站，你的自动构建和测试机就是一个理想的候选。如果做不到，至少要有一个开发人员每天在这个环境中工作，这样可以很快捕捉到错误。

技巧 21

如果生产环境会用到，你也要用到

开始增加功能之前一定要保证你的系统能运行。与构建这个中空的外壳相比，增加功能会花费更多开发时间。在投入时间让这个中空外壳做具体的工作之前，一定要让它首先能够完成端到端的工作流程。

现在项目已经具备：

- 一个完整的、有文档的架构；
- 一个表明架构能够正常工作的概念验证。可以做一个客户调用来确定它能端到端运行；
- 团队之间有清晰的界限；
- 产品功能域之间有清晰的界限；
- 与负责相邻代码层的团队有过会面。让所有团队相互交流有很大好处。

在桩中填入功能代码

现在已经有了一个端到端的工作系统。每个部分可以相互通信，而且所有代码都能编译和运行。现在可以让它做具体的工作了。

就像“好篱笆促成好邻居”[①]一样，好的接口也可以促进团队交互。如果必要的话，现在每个团队都可以完全隔离地工作，开始填入各个接口底层的逻辑。每个团队现在有了一个基本框架，他们可以开始填入代码。只要不破坏现有的接口，你可以做你想做的任何事情。没有人能改变层之间的API，除非相邻层的两个团队都同意。

你现在可能遇到的诱惑是想要开始增加简单的功能，比如让 Login 真正起作用。要克制住这个冲动！在产品真正交付那一刻之前，有一个正常工作的 Login 将是一种“奢侈”。为客户演示时并不需要它，而且它会妨碍你完成其他开发工作。不过，让核心功能运转不算奢侈。如果核心功能不能正常实验，就不能算做出了一个产品。

开始实现功能时，先要针对包含新技术的领域，本质上很困难的领域，或者对你的产品来说核心的领域。如果你的项目在使用一个第三方制图软件包，此前没有人用过，你就应该先集成这个软件包。如果数据分析例程是产品的核心，就要先编写这些例程，让它们能正常工作。换句话说，要尽早解决最难的问题，把容易的留到以后解决。

① 这个成语的本意是“保持距离，友谊常青”，这里作者使用了这句话的字面意思。——译者注

技巧 22

先解决最难的问题

如果你遇到需要额外花费时间来解决的问题，或者需要更换组件，要尽早发现这些问题。不要等到最后一周才发现你根本无法处理一百万个数据点，或者你的部件不能像原先希望的那样工作。尽早发现这些问题，然后修正问题或者调整时间表。如果你在使用“任务清单”（见实践 10），可以删除一些低优先级的任务项来弥补损失的时间。不过，如果你把最困难的问题留到最后，就根本没有重新规划的余地。

较早注意到棘手的问题还可以尽早从项目时间表中排除风险。这有一个很好的作用，可以确保项目最后的工作很容易。就在大多数团队还在为困难的部分一筹莫展时，你的团队已然在悠哉悠哉地实现 Login。

这个阶段也很适合发现潜在的性能问题，特别是与网络流量有关的问题。开始填入代码时，要特别关注耗时很长的功能。例如在网络上查找 CORBA 对象的功能就属于这一类。如果你很少使用这种例程，它们应该不会导致问题。不过，如果频繁地调用这些功能，你就会发现项目的性能极差，运行缓慢如牛。如果一段代码在你的开发环境中就运行得很慢，在客户的大负荷服务器上就更没有机会快速运行了。

重构和求精

在一个实际项目中，随着项目被不断明确和更好地理解，层之间的接口自然会改变和演进。这些不断进行的改进是不难想见的，代码的演进也很正常①。不过修改过程不会很费劲：你的团队已经在合作，而且你可能已经建立了不错的人际关系。

很多过程方法会把项目分解为阶段，就像原来的瀑布开发模型。瀑布开发有一组非常严格的阶段，一个接着另一个：需求、规范、设计、实现、测试和维护。当然，像这样使用阶段有一个前提假设，认为你早在开始之前就已经对这个问题完全了解，而且可以准确地规划每一个阶段。对于任何一个大型软件项目，如果你认为团队在开始之前就能充分了解项目的每一个细节，

① Jim Highsmith 经过观察认为，重要的是在最后一刻而不是第一时间保证正确。

是很可笑的。从我们的经验来看，分阶段的项目通常都无法赶上最后期限交付产品。

相反，需要在整个软件开发周期中保持灵活性，而不是试图强制项目的时间表满足预定的阶段。团队为桩代码增加实际的功能时，可能会发现某个接口需要传入一个额外的变量，或者完全漏掉了一个接口。可以在任何时间增加这个功能。过程方法的关键是允许你在需要时做出这些增量修改，而不是看进度是否允许。

做出这些修改时，要公开发布，免得使用同样接口的其他团队诧异。如果需要，可以另外增加一个接受额外变量的新接口，而不必删除其他团队正在使用的某个接口。要记住，曳光弹项目中绝对不要破坏构建。增加的代码只能扩展系统，而不能破坏系统。

破窗理论

一旦系统运行，就不允许任何人破坏。

原来的“桩”系统在项目的生命期中继续运行。随着你为特定接口编写代码，功能会扩展，不过其他接口还应当继续运行。要把对产品的任何破坏都当作“打破的窗户”（见[HT00]中的描述）。一旦破坏了一个功能，就很容易破坏第二个、第三个甚至第四个。不过，没有人希望成为破坏正常系统的第一人。因为你的曳光弹系统已经启动，很早就在运行，所以产品每天都会保持干净，蔓延问题不会积累下来。

还要认识到，有时需要丢掉你原来编写的代码。比如这个代码运行太慢，或者它返回的结果是错误的。智者千虑，必有一失。可以完全重构或重写代码，只要接口仍以同样的方式工作就可以。在一个曳光弹开发项目中，可以在任何时刻做这些修改。但有一个原则：如果没有达成共识，就不能修改其他团队正在使用的接口，不过你可以根据需要修改这些例程底层的具体代码。只要接口采用同样的方式，其他团队就不会知道你已经修改了代码。他们的代码仍以同样的方式使用你的接口，只不过现在运行得更快了，或者可以返回正确的数据了。

一个简要的例子

我们曾经参与过一个项目，展示出这种分层封装在架构方面的显著优点。

这个项目包含 4 个系统对象：一个客户层、一个 Web 服务器层、一个数据处理层，和一个数据库访问层（见图 4-1）。

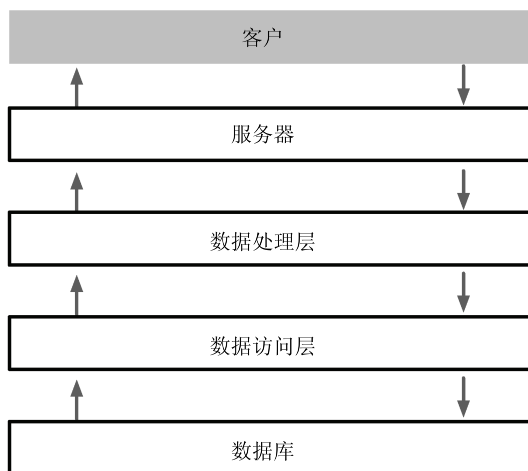


图 4-1　包括 4 个主要层的系统

这是为一家生物技术公司开发的项目。客户应用允许用户上传一组分组的数据点（一个模块取药物分子数据，另一个取 DNA 片段数据，等等），系统会查找这些数据点的共同特点。数据库包含实验结果、从学术出版物中挖掘的信息、多个公共生物学数据库以及其他资源。应该说数据库相当复杂。鉴于数据连接的方式，要找出上传数据之间的关系需要大量数据库访问和数据处理。

如果上传了 10 个 DNA 片段，这个系统就要查找各个片段之间的联系。有多少在相同的科学文献中提到？有多少出现在鼠类或人类相同的 DNA 链中？文章中有没有提到更大的 DNA 链？有没有其他实验涉及这些 DNA 链？是否以某种形式公开发表过不同数据点之间的已知关系？通过几个步骤找寻关系之后，要检查的数据点数目会增长好几个数量级。然后运行各个统计分析例程来查找关系，或者对结果生成图表。

这个系统相当直接。客户应用上传请求。用户向 Web 服务器上传信息之后，Web 服务器会把数据分组，并把分组的信息发送给数据处理层。数据处理层向数据层请求各个数据点的有关信息，然后开始分析结果。Web 服务器会比较所有结果，并发回给客户。

向这个项目增加特性时，我们注意到数据库层变成了一个瓶颈。查询运行得太慢。我们优化了数据库调用，但是没有改变数据处理层使用的API。这些

修改对数据处理层是透明的。数据处理层仍以同样的方式请求数据，但是数据访问层会更快地返回数据。

不过，随着项目需求继续改变，我们需要支持原先不曾想像的一个更大的数据吞吐量。我们采用了一种非常基础的方法：将底层数据库调用分布到一组数据库服务器上。我们在不同机器上并行地运行客户查询（见图 4-2）。系统性能随所用的数据库机器数线性增长。

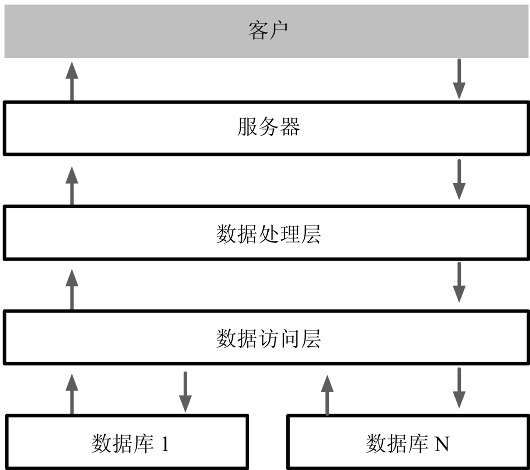


图 4-2　有多个数据库的 4 层系统

大多数项目中，实现一个并行数据库查询机制都需要对整个代码基做重大重写。不过，由于我们使用了曳光弹式清晰定义的层间接口，因此可以完成这些重大修改而不影响上面的层。在这里，数据挖掘层根本没有任何改变。我们保证数据库访问层 API 不变，而重写了接口内部的代码。新代码能够把数据库调用分布到多个服务器上。从数据处理层的角度看，只是一切都会更快地运行而已。

后来我们发现数据处理层成为了又一个瓶颈。数据层提供信息的速度太快，数据处理服务器跟不上了。这样一来，数据处理层就成为了瓶颈。

幸运的是，采用这种体系结构时，可以把工作并行分布到多个数据处理服务器上。我们增加了更多的数据处理服务器，让 Web 服务器层把到来的请求划分为组。每个组分别发送给一个不同的数据处理服务器（见图 4-3）。

图 4-3　有多个数据处理服务器的 4 层系统

尽管我们在分析非常庞大的数据集，但是各个组的分析相互独立。由于使用了相同的接口，而且只提供少量的数据来进行分析，数据处理层并不知道有任何改变。我们可以向系统增加任意数量的数据处理服务器，可以根据客户的需要处理更大的工作负载。

当然并不是对任何应用都可以完成这种重构和并行化处理，不过这个例子展示了在不同层之间清晰地定义接口所带来的好处，你会得到出色的灵活性和强大能力。我们可以在一层中做重大调整而不影响相邻的层。

另一个要说明的重要问题是，尽管这个例子谈到使用不同的服务器，但并不要求你一定这么做。在这里我们使用不同的服务器只是为了展示功能划分。这个原则同样适用于应用。以模型–视图–控制器（MVC）模式[①]为例，清楚地定义逻辑、数据和图形用户界面（GUI）之间的接口，就能把各个组解放出来，让他们独立工作。实际上，使用 MVC 的目的就是为了实现这种封装。

技巧 23

封装的架构是一个可伸缩的架构

推销曳光弹

曳光弹开发有巨大的好处。不过，要向你的工作室引入这个思想，可能需

① MVC 概念是一个久经考验的封装模型。它要求在数据视图、数据模型和控制器（或逻辑）之间清晰分界。这个概念在这里也适用。如果你还不熟悉 MVC，现在就该去了解！

要向你的团队成员或老板推销这个概念。考虑到这一点，我们在这里总结了采用曳光弹模式工作的好处。

团队可以并行工作。一旦建立接口桩，团队就能独立工作。

你的客户能更早看到“工作的”系统，并更快提供反馈。团队一旦建立他们的接口桩，你就能展示系统。增加功能时，新特性可以立即展示。要记住，客户越早看到产品，就能越快地修正产品方向中存在的问题。通常客户对自己想要什么有些想法，但是不能为你清楚地定义需求。这类客户需要看到一个真正运行的系统，才能对产品的方向提供切实的反馈。采用曳光弹开发，你可以很快创建一个运行的系统，而不用编写稍后就丢弃的演示代码。

向前推进！千万不要低估士气对生产力的影响。如果每个人都完成了工作，产品就要在大家的努力之下“出炉”，此时士气肯定会高涨。

一旦定义了接口，测试团队就可以开始编写在接口上运行的自动测试脚本。在真正测试完整的产品之前（也就是在所有部分都已经填入代码，并且可以正常运行之前），测试团队应该已经有了一组可以运行的自动测试。这些测试会随着接口的改变而改变，但是框架一直不变。尽管有修改，但是不会从零开始。有了这种测试套件，就可以独立地验证系统中的每一层。

采用曳光弹模式构建一个项目时，你要建立一组团队，并为他们的交互建立一些基本原则。应该要求他们互动，并对项目的关键问题达成一致。除了学习与其他团队沟通，还要允许团队定义接口修改，这会给每个团队更大的产品归属感。他们原先定义了项目接口，已经很有归属感了。现在他们要对接口进行改进和提高。因此，每个人都会更努力地工作，希望他们的项目成功。

团队采用这种方式并行工作可以得到巨大的生产收益，这样可以非常高效地使用你的开发和测试时间。我们曾经参与过只有4个开发人员的团队，我们一同为每一层定义接口，然后4个团队成员分别单独编写某一层。一旦写好了桩，我们就有了一个可以运行的系统。

产品不能正常工作时，你就该知道肯定有人破坏了某个接口，从而很快修正问题。大团队也可以使用这个方法，同样可以取得成功。基于这种自治性，团队成员可以很容易地从一层移到另一层。团队成员已经对相邻层需要做什么有了一个基本的了解，因为他们曾经帮助定义接口。如果某个方面落后，就可以很容易地让开发人员从一层移到另一层。

这种开发方法一旦使用，其效果会立竿见影。一旦你有一个曳光弹，提供了一个 GUI，客户就可以看到它是什么样子，并开始向你提供反馈。有足够多可以正常工作的功能时，你就可以决定交付产品或者完成 beta 测试。销售团队不必等到“所有功能”都完成才开始演示，因为你已经有一个端到端的工作系统。还可以从客户反馈中了解下一步需要完成哪些特性。你的开发团队不用猜测优先级，因为这将由“客户驱动”[①]，即由客户帮助确定。让客户加入到开发周期中会带来巨大的好处，不论你在一个小型公司还是一家大型公司。大多数情况下，如果客户不满意，所有人都没有好日子过。

如果船没有移动，转动方向盘是没有用的。对于开发和测试团队来说也是如此。如果每个人都在移动，保持继续移动就会容易得多。采用 TBD 会让你的所有团队朝着同一个目标前进，同时保证各个团队可以独立工作。每个团队自行工作，但是会朝着所有成员共同定义的同一个目标努力。如果团队在前进而且势头迅猛，团队成员就会看到一直以来取得的进展。每个团队成员都在为一个实际工作的系统做出贡献，它每天、每周都在改进。你今天增加的代码就会影响现在运行的系统。“推进”对士气的影响非常惊人。

技巧 24

除非船在移动，否则转动方向盘没有用

曳光弹开发过程可能并不适用于所有情况（不过我们还没有见过不适用的情况）。这是管理项目的一种非常有效的方法，可以扩展到非常庞大的团队，处理任何规模的项目，为架构提供灵活性来完成惊人的工作。曳光弹开发具有功能领域的封装、并行工作、快速的客户反馈以及很多其他特性，这使它成为较为灵活和有效的软件开发方法之一，你的下一个项目确实应当尝试这种方法。这是一个非常强大而且用途广泛的工具，应该把它收集在你的工具箱里[②]。

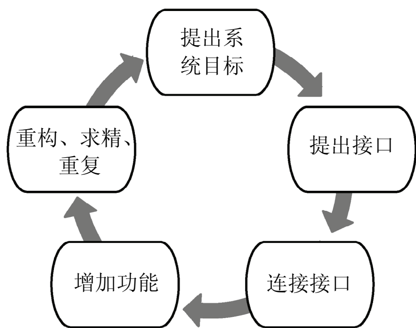


图 4-4　曳光弹开发

① 说真的，这不只是一个时髦用语。

② 要记住，把一个工具放在你的工具箱里并不表示每天都必须使用它。只是说如果需要，这个工具确实可供使用。

如何起步

要开始使用 TBD，最好的方法是选择一个项目开始尝试。开始之前，先与你的团队成员回顾一下有关概念。我们建议你先从一个较小的项目开始尝试，感受一下 TBD 是如何工作的。

- 定义系统对象。
- 定义系统对象间的接口。
- 编写接口桩。
- 实现桩之间的相互通信。
- 在桩中填入功能代码。

做到这些说明你做得对

- 整个系统一直可以“运行”。
- 团队成员除了了解他们的对象，还了解系统对象。
- 团队成员可以轻松地帮助完成其他系统对象。
- 可以重写代码基中大量的代码，但不会造成破坏。

警告信号

- 团队成员针对设计一个接口的“正确方法”争论了几个小时。
- 这个端对端系统从来不能真正地端对端运行。
- 构建经常被破坏，因为接口改变了但客户不知道。
- 只有一个无所不包的庞大的系统对象。
- 有 700 个细小的系统对象。
- 团队已经工作了几个月，但系统还是没有能够编译的桩。
- 团队已经做了几个月编译桩的工作，但还是无法提供完成的特性。

大多数人在回避问题上花费的时间和精力比他们尝试解决问题所下的功夫还要多。

►亨利·福特

第5章 常见问题及解决办法

读到这里，你可能在想："这些都很好，但是在真实生活中会怎么样呢？我到底能不能用这些方法来解决我当前的问题？"为了回答这个问题，我们专门为你写了这一章！

这一章会简要介绍我们自己遇到过的一些常见的问题。每一节首先会描述这个问题，然后简要介绍我们的解决方案。必要时，我们还会更为深入，针对你的特定角色（例如，开发人员、技术领导人、经理和客户）特别定制解决方案。当然，并不是说你不必读对应其他角色的小节。通过读这些内容，你会对"另一半"有所了解。你还会了解到另外一些技术，可以对这些技术加以调整来适应你的角色。

例如，即使你不承担一个管理层角色，也可以通过建议、讨论和劝告的方式应用管理层解决方案。（当然真正的管理层也可以这么做！）

你会注意到，这一章所有小节都有一个共同的基本主题：行动起来。这一章并不适合那些满足现状或放弃战斗的人。不过如果你想面对挑战，渴望做得更好，就请继续读下去。

16 救命！我继承了遗留代码

你继承了一个遗留产品，要对它进行维护和改进。掌控这个产品最快的方法是什么？那就是构建它，使其自动化，最后测试之。

(1) 构建：首先明确如何构建，然后编写脚本完成构建过程。这个任务并不一定容易，特别是当代码仅由一人所有时。这种代码通常只在一个机器上构建，因为它要依赖于周围的环境。完成这一步后，任何人在任何机器上都可以构建这个产品。之后，自动化构建就会很容易了。

(2) 自动化：你的目标是在一个干净的机器上自动构建和测试整个产品，并尽量不去人工干预。我们并不是说完全没有人工干预，这里需要有所权衡。有时手动地安装和配置环境中的一个支持部分会比编写脚本自动完成更为容易。只安装一次的应用就可以手动安装（如编译器等）。记录所有构建步骤，并公开这些文档。

(3) 测试：明确代码做什么，然后为它编写模拟客户测试，并开始测试。一旦干净利落地完成了项目构建，你就会希望确认它能正常工作。为了编写这些测试，你必须清楚地了解这个产品要做什么（这并不奇怪）。

模拟客户测试是个不错的起点：可以用来测试产品的一般功能，因为它们就相当于一个产品用户。

(4) 更多测试：明确产品的内部特征（如结构、控制流、性能和可伸缩性等），编写更多测试。除非产品无可救药，根本不可用，否则总会有需要修正（或至少要记录）的 bug，或者必须做的改进。对遗留代码做这些修改通常是一件非常可怕的事情，因为你永远不清楚做出一个代码修改时会影响到什么。不过现在你可以放心地去做，因为你编写的模拟客户测试就相当于一个安全网，可以保证你不会造成太大的破坏（你确实写了这些测试，是不是？）

要对修正的每一个 bug 和增加的每一个改进编写一个新测试。编写什么类型的测试取决于做了什么修改（例如，单元测试用于测试一个底层内部变更，模拟客户测试则用于测试一个新特性）。在这里对遗留产品的处理与你支持的任何其他产品并没有不同。

完成这些工作之后，任何人都能支持这个代码！他们能够在任何机器上自动完成产品构建，然后在他们的桌面计算机上运行自动测试来确保产品能

正常工作。自动构建系统会在一个干净的环境中运行构建并再次测试来确保一切正常。

技巧 25

测试之前不要修改遗留代码

17 测试不可测试的代码

如果你是一个开发人员或测试人员，想要为产品建立一个自动测试套件，可能会发现事情很难办：你想测试一个产品，而它没有适合你的测试鱼杆的“挂钩”。测试“不可测试”的代码并非不可能，不过创建和维护这些测试很费劲，往往会抵消自动化带来的好处。

例如，你可能想测试一个 HTML 页面，其中包含大量未命名的域。所以，不能使用 name 或 id 标记来查找一个域，你必须统计页面上的项数，检查（比如说）第 15 项中的值。这会让别人理解和维护起来都非常困难，而且测试本身也很脆弱。另外对页面的修改往往会破坏你的测试。

有什么解决方法呢？告诉经理你希望启动“测试驱动重构”。你想为应用增加一些简单的挂钩，从而可以（或适合）顺利完成自动测试。

首先，为产品创建（或借用）一个测试计划。开始时要力求计划简单。不要期望第一次就十全十美。先对基本功能完成一遍测试。

其次，编写一个自动测试。让测试尽可能向前推进。如果“卡住”，最少增加多少产品代码才能让测试继续运行？是不是只需要向现有 API 增加一个返回码，或者为一个 HTML 域增加 id 标记？

不要试图第一次测试就修正每一个问题，也不要试图向产品中的每一个页面都增加 id 标记，而要针对能让测试通过的那一个页面。如果测试计划遇到阻碍，不要停下来。把不能测试的那部分测试计划先去掉，继续前进。要记住，这个难测试的部分还可以增加到下一个测试计划中。

这里的目标是增量改进和保持前进。当你完成这些小改进时，开发人员就会开始了解如何创建可测试的产品。另外你在编写自动测试时也能学到技巧。你会惊讶地发现，一旦有了一个基本的自动测试套件，居然就会得到这么多支持。

技巧 26

使用测试驱动重构清理不可测试的代码

18 特性不断破坏

下面这种场景在很多开发工作室里都经常出现：你的团队向测试部门发送了产品的一个临时版本。第二周你收到一个测试人员的来电，告诉你产品的一个主要特性再次被破坏。在之前的 7 个内部版本中，这种情况已经是第 5 次发生了。不过这还算“幸运”，起码测试人员是在产品交给客户之前发现了问题。上一次则没有这么幸运，产品交给客户后才发现问题，结果客户不得不退回到产品的前一个版本，而你差一点失去这个客户。

查看了这个特性的代码后，你发现上个月你做的 bug 修正莫名其妙地不见了。这一天接下来的时间你只好重新输入那些 bug 修正代码，晚上也用来加班，而不能去按原计划做其他的事情。

修正这个问题最快的办法是什么？就是增加一个自动测试套件（见实践 7）。要让产品或平台尽快稳定，模拟客户测试套件是最佳的选择。模拟客户测试能在最少的时间内测试最多的代码行。

技巧 27

模拟客户测试可以事半功倍

只要代码修改就自动运行测试套件，这样一个特性破坏时你就能立刻知道（见实践 4）。一旦重新引入 bug，这个系统会立即将其捕获并通知负责的开发人员。这样修正问题的责任就可以交给做出这个修改的开发人员了。

如果不创建一个自动测试套件，那么你只能依靠手动测试来查找 bug。这样不仅很慢，而且容易出错。如果不自动地运行测试，就意味着你在积累潜在的破坏，幻想能将它们集中修正。

要保证你的产品稳定，就要利用自动模拟客户测试来进行测试，而且每次代码修改后都要运行这些测试。

19 测试？我们早就不用了

你的团队投入了很多时间来创建一组可以自动运行的测试，但是这些测试并没有得到使用。

也许你根本没有测试团队，而开发人员太忙所以无暇顾及。或者尽管你有测试人员，但是他们无法运行这些测试，因为他们的机器上没有安装正确的环境。渐渐地，测试出现了问题，但是没有人意识到这是因为没有人运行它们。等到有人想要运行测试时，结果已经没有意义了，因为测试与代码基已经不再一致。测试变得毫无用处。

如果发生这种情况，就该重新开始了。没有人使用测试可能只是因为他们从测试中得不到好处。一般说来，好处很直接也很明显，但是开始（或重新开始）时总会有一个前期开销，你必须承受得了。毕竟，诺亚造方舟时并没有下雨。

另一个要面对的问题出现在项目生命期的后期。增加的测试越多，编译之后运行这些测试花费的时间就越多。达到某一点时，可能无法每次都运行所有测试。应当选择哪些测试继续运行呢？你选择的测试应该能充分地测试那些正在积极开发的代码。如果你注意到一个测试在每晚或每周构建中失败，就要把它移到 CI 测试套件中，更频繁地运行。我们在“使用自动化测试框架”中（见实践 7）较为详细地讨论了这种针对性测试。

技巧 28

持续测试不断改变的代码

如果需要，应当在不止一个机器上运行 CI 系统，从而能并行地运行测试。采用这种方法，一个需要一小时的测试在 4 台机器上并行地运行，15 分钟就可以完成。这也是一种得到跨平台测试覆盖率的好方法。

我们工作过的一个小型创业公司里，Windows、Linux 和 Solaris 上都运行了同一个 CI 系统。每个平台都运行同样的测试套件，从而确保产品在各个平台上都能运行。尽管开发人员主要在一个平台上开发，但我们会在每一个将要交付产品的平台上不断进行测试。

你也应该这样做，这样可以避免到发布周期的最后才暴露出问题，需要修正特定于平台的一些 bug。

除非你有了一个 CI 环境，能够在每次代码变更之后构建产品，否则对代码做的测试就不算多。甚至每天构建都不够。可以这样来看：一个开发团队一天中可能会修改大量代码。团队运行测试套件时，他们会查看测试结果，隔离出问题，然后找出哪些代码变更导致了这些问题。如果只有几处代码变更，这会很容易。团队中几乎任何人都可以做这个工作。

不过，如果你在修改了大量代码之后才进行构建和测试，这就会变得困难得多。如果你等待的时间太长，最终只有对产品、产品代码和测试都极其熟悉的人才能够将失败的测试与代码对应起来。这个任务往往要落在一个高级团队成员身上。这样一来，他们就无法增加特性或修正bug，而只能跟踪其他开发人员的代码。

这绝不是高效的做法，也正是由于这个原因，一个好的 CI 系统可以提高整个团队的生产效率。

20 不过我这里没问题！

你有没有遇到过这种情况，你报告了一个bug，收到的回应却是："我这里没问题！"更糟糕的是，你曾经收到过一个bug报告，试图再生这个bug失败后，你自己也做出这样的回应？说实话，没有人关心在你那里有没有问题。你的客户、测试人员以及所有使用产品的人只关心他们自己是否可以正常使用。如果有人报告了bug，千万不要止步于"我这里没有问题"。要知道，对别人来说肯定是有问题的，否则他们根本不会报告这个bug。

如果在你自己的机器上无法再生某个bug，可以考虑在构建机上试试，这可以作为你的保险。用它来验证在你的工作站上无法再生的bug。一旦在构建机上重复了这个bug，再查找你的系统有什么不同。也许你已经修正了这个bug！或者可能必须向交付产品（或其环境）增加些什么，使产品在任何地方都能正常工作。一旦重复出bug，就要创建一个测试来暴露这个bug，然后在干净的机器上运行这个测试。如果仔细编写测试，甚至可以把它发送到现场，看看客户是否可以重复你的结果。

当然，如果在构建机上也不能再生这个bug，就可能是客户自己的机器上存在配置问题。可以对他们建立的环境问一些问题。发现导致bug的不同配置后，你可以修改代码来支持这个差别，或者柔和但坚定地告诉用户他们需要调整配置。

技巧 29

必须适用于所有人

21 集成代码很痛苦

你是不是非常讨厌编译产品的代码基？从源代码管理系统更新你的本地代码树之后，你是不是要花几个小时才能让它与其他所有人的修改一致？你是不是想“保护”你的机器上的代码，而只在无计可施的情况下才会考虑合并团队的其余代码？你的团队是不是把“代码集成”看作是一个贬义词？可悲的是，并不只是你一个人这么想。

我们工作过的一个工作室就是这个问题的典型代表。他们只有十几个开发人员，但是团队成员工作好几个月都不提交也不更新代码。如果一个开发人员不得不构建产品，他首先会签出代码树，试图编译。编译失败时，这个倒霉的开发人员（暂且把他叫做 Hal）会打印出编译错误，力图找出这些出问题的地方由谁负责。可怜的 Hal 会逐个地造访各个开发人员，请他们修正相应的编译问题。每个开发人员修正一个问题之后，Hal 会更新他的代码树并再次尝试。每个开发人员完成这样一轮过程需要花费大约半天的时间。所以，如果有 12 个开发人员，Hal 往往需要花费一周的时间，才能得到能够正常编译的代码。等到 Hal 的所有问题都解决时，也许另一个开发人员又提交了不好的代码，Hal 只好重新开始。

怎么才能阻止代码集成变成一个持续不断的恶梦呢？

你会遇到两类问题。两个团队成员在编译同一个文件或者相互依赖的几个文件时，会出现第一种问题，其结果是无法编译。方法签名发生改变就是这种情况的一个典型例子。方法 loginUser 原来只有一个字符串参数，现在它需要两个参数。所有使用原签名的代码都将无法编译。

第二个问题更阴险——代码能编译，但是不能正常工作。例如，Joe 认为整数变量 SystemStatus 应该包含一个返回码，而 Cathy 使用 SystemStatus 来跟踪程序哪一部分是活动的。Joe 和 Cathy 的代码以不兼容的方式使用了同一个变量 SystemStatus，如果他们提交了代码，两人的代码就都无法正常工作。

等待集成代码的时间越长，出现这些冲突的可能性就越大。你等的时间越长，你（和其他团队成员）编写的代码就越多。随着编写的代码行增加，冲突也会增加。冲突越多，集成就越痛苦。解决办法很简单：更频繁地集成代码，从而减少冲突，简化合并工作。

要使用 CI 系统（见实践 4）。CI 系统会在每次代码提交之后编译产品的所有代码，并且在成功构建之后自动测试产品。所以完全无需你的干预就可以轻轻松松地捕获这两种集成问题了。（这样还可以方便技术领导人检查哪些开发人员一段时间内没有集成代码。）

技巧 30

经常集成，并持续构建和测试

22 不能可靠地构建产品

我们曾经在这样一个工作室工作过，他们要对产品连续构建 3～5 次才能得到一个好的构建。这种状况已经持续很长时间了，开发团队便习以为常。完成构建时，会有一个开发人员运行一系列脚本（由 IDE 生成），中间还夹杂着一些手动步骤。没有人了解这些构建脚本做了什么。对此，他们并不是想办法搞清楚，而只是找借口，声称“我们的产品太复杂了”。

我们花了半天时间创建了一个完全自动的脚本，可以从头构建整个产品。除非代码不能编译，否则总能完成构建。所有人都很震惊！不过这确实没有那么难。我们首先假设构建可以自动完成，然后针对他们所用的语言找到一个适合的脚本工具。这个工具提供了足够的示例代码段，我们只需编写很少的原始代码就能让它正常工作。

为什么你的构建不可靠？可能过程太过复杂，以至于很容易漏掉某个步骤，或很容易以不正确的顺序执行步骤，或者构建过程中可能存在循环依赖性[①]。为构建过程建立脚本可以解决这个问题，因为脚本会以一致的方式执行步骤。一旦有了脚本，每次就会采用同样的方式完成构建。不能可靠地构建产品可能还有一个不太明显的原因：你没有清楚地了解你的构建过程。通过创建一个构建脚本，你就能够准确地了解所有构建步骤。

除了提高可靠性，自动构建还可以为产品提供专业“抛光”，使新团队成员的转移变得容易，而且使你能轻松地使用 CI 系统。有关的更多好处和提示见实践 4，也可以参考《程序员修炼之道》中关于“无处不在的自动化”的一节 [HT00]。

① A 依赖于 B，而 B 也依赖于 A，需要好几轮才能发现。这通常说明你的代码中存在一个设计缺陷。

23 客户不满意



小乔爱问……

如果我没有客户呢?

如果你不是一个经理或团队领导人，也不要以为这一节不适合你。不论开发人员与外部世界多么隔绝，他们也同样有客户，那就是他们的经理。你不想让你的经理成为你的狂热粉丝吗？可以把他们看作是你的客户。

要想很好地管理你的客户，就要知道他们想要什么，然后交付他们真正想要的产品。如果他们变成你的狂热粉丝，你就大功告成了。很简单，是不是？不过，要明确客户真正想要什么，往往是项目中最困难的部分。

要让客户与你并肩工作，帮助你定义产品，同时对他们的期望加以管理，明确哪些可以达到而哪些不可能。不要只是在项目开始时接收客户的输入，而是整个开发过程中一直要与客户保持联系。客户反馈能保证你不会构建出没有人想要的错误产品。这也会让客户感觉自己是这个产品的共同所有人。产品会从“你们的”变成“我们的”。最后你会得到一个合作伙伴，而不是一个吹毛求疵的批评家。

那么如何与客户交流项目的状态？最好的办法之一就是创建一个真实系统，可以尽早演示，甚至在特性集还不完备的情况下就可以演示。不论产品有一个 GUI 或者只是一组应用编程接口（API），都可以采用这种方法。为了在开发周期中尽早提供一个真实系统，可以使用曳光弹开发（见第 4 章）。团队每完成一个新特性，就把它增加到演示程序中，让客户能够看到并给出反馈。

技巧 31

尽早而且经常发布真实演示系统

发布演示系统的间隙如何与客户沟通？要鼓励客户尽可能经常检查“任务清单”。保证“任务清单”是最新的，让客户能够看出它有所改变。还要邀请客户加入你的开发世界，使他们能够看到你的开发方向和存在的问题。如果客户了解是什么原因导致的项目延迟，往往就更能接受延迟状态。另外，要鼓励客户对“任务清单”提出修改建议。如果他们不喜欢项目的走向，应当要求你做出调整。毕竟，最后付账的是客户。如果客户经常与你交流他们的需求，而且你的团队不断调整产品来满足他们的需求，那么最终结果肯定是皆大欢喜。

24 有一个另类的开发人员

每个工作室都至少有一个看起来总是与团队其他人步调不一致的开发人员。尽管他经常造成破坏，却总是坚信自己是对的，其他人都是错的。如何能让这匹脱缰的野马走上正轨，真正有工作成效呢？

这个问题只能由经理来解决。如果你是一个开发人员，管束同事绝不是你的任务（而且如果你们为此在停车场打一架，仲裁结果多半并不会偏向你）。如果你是一个客户，你可以要求特性和稳定性，但不应以个人身份与个别开发人员交涉。这个问题要留给开发经理来解决。

作为经理

如果你是经理，可以采取很多措施“收服”这些另类的开发人员而不至于把他们赶出团队。下面来看可以采用哪些方法充分发挥他们的能量。

- 每日例会可以让这些开发人员保持高度责任感（见实践 12），而不至于偏离太远。通过团队每天会面，检查大家都在做些什么，你可以每天修正他们的“航向”。虽然还是有可能损失这个人半天的工作，但是总不至于损失一周、一个月或者一个季度的工作。每天这个另类开发人员必须报告他做了什么，他打算做什么，有哪些问题。如果你注意到他实际做的与头一天计划要做的工作不符，就要在每日例会中提出来。

  我们曾经与一个酷爱重构代码的开发人员共事，他虽然不太了解代码的功能，却也热情洋溢地要去重构代码！完成了分配给他的任务后，他就会着手重构在工作中偶然遇到的任何代码。例如，如果他调用了一个类，这个类包含一个统计算法，他就会对它重构，尽管他根本不清楚这个代码要做什么，以及为什么这么做。他本来希望“整理”代码，遗憾的是，结果却往往破坏了代码。他总是调整代码的格式，让它看起来美观，但是最后这些代码通常不能再像以前那样运行了。只要系统中出现随机的 bug，我们就会检查源代码存储库历史，来看有没有这个人的工作记录，往往会发现他确实做过手脚。

  意识到这种情况后，我们便确保在每日例会中为他分配足够多的工作，让他一直工作到第二天。有时，他还是会提前完成分配的工作，又开始四处巡视代码，但是总的来说，每日例会就足以把他“收服”。只需确

保每天给他分配足够充实的工作量，让他很忙而无暇顾及其他。

一般来讲，另类开发人员并不是恶意的。他们只是想帮忙，但超出了你要求的范围。只需要更多的指导和监督通常就能解决问题。不需要对每一个团队成员都实施这种监管，这只针对那些容易信马由缰的人。

- 使用“任务清单”也是管束另类开发人员的一个好方法（见实践 10）。如果每个人都使用自己的任务清单来推动工作，他们就会发现，自己没有足够理由去做任务清单上没有的一个任务。作为他们的经理，如果你能帮助填写每个团队成员的任务清单，就会达到最好的效果。

  另类开发人员通常只是想帮忙。我承认，他们帮忙的动机往往很独特（可能与你的不同）。使用任务清单，可以让这些另类开发人员清楚地知道你认为哪些任务更重要。

  不必限制另类开发人员只能完成产品特性有关的任务。除了让他们高负荷地工作，完全也可以让他们不时地“轻松一下”。如果他们想重构，可以在非生产代码上练练手。或者可以让他们发明某个新工具，研究一个新的编程技术，再传授给团队。不过一定要把这些任务放在任务清单上，使他们有一种责任感。

  有时你会发现一些另类开发人员非常自以为是，根本不关心给他们分配了什么任务。他们认为经理或技术领导人无足轻重，没有他们懂得多。不论你多么清楚地布置工作路线，他们也视而不见。在这种情况下，只有一件事可以做：记录这些另类开发人员的所作所为，然后以不服管理为由将其辞退。在大多数情况下，只要开始这个记录过程，就会让另类开发人员偃旗息鼓，气焰全无。如果他们依然我行我素，那么最好还是把他们请出团队。

- 代码审查也是一种掌控另类开发人员的好方法（见实践 13）。审查另类开发人员的代码时，他们必须向你解释他们做了什么，以及为什么那么做。他们不能再悄无声息地提交代码（可能根本没有人注意到）。你会查看每一页（我们希望是每一行）代码。如果你发现他们的工作与任务清单上的任何工作都没有直接的关联，就不要批准代码提交。如果有一个特别固执而强势的开发人员，就让一个高级团队成员来完成代码审查，而不能是随便某个人（否则可能受其威逼不得不批准一个糟糕的代码变更）。

- 使用自动代码变更通知（见实践 14）。你可以准确地检查提交了什么代码，以及出于什么原因提交这个代码。查找这些另类开发人员的有关通知，看看他们处理的文件或包是否与分配给他们的任务毫无关系。
- 开始使用 CI 系统（见实践 4）。另类开发人员即使签入了尚未审查的代码，或者没有通过邮件向大家发送一个完整的代码差异部分，也无法躲过虚拟构建监视器的法眼！每次有人签入代码时，CI 系统就会构建项目，运行测试，然后向项目联络人发送邮件，提供一个有关文件修改情况的报告。如果另类开发人员修改了本不该修改的文件，你就会马上知道。

  使用 CI 系统来监视团队成员应该作为最后一道防线。这往往不是使用 CI 系统的初衷，不过如果确实有一些另类开发人员总在破坏代码，就很有必要这么做。CI 系统可以解决大量问题，不过这可能是其中比较极端的一个。

可以看到，你有很多工具来对付另类开发人员。使用这些工具，大多数情况下让他们走上正轨都很简单。最终，他们会养成好的工作习惯，你也不用再那么严密地监视他们了。

总结一下，你可以采取以下措施：

- 使用每日例会修正另类开发人员的航向；
- 保证另类开发人员只能完成任务清单上的任务；
- 使用代码审查和自动代码变更通知来跟踪另类开发人员的工作；
- 使用 CI 作为最后一道防线监视另类开发人员的工作。

25 你的经理不满意

怎么让你的经理满意？可以用一个词来概括，那就是沟通。一定要让你的经理总能了解你在做什么，以及为什么这么做。这会不会影响生产效率呢？你可能会问。也许你是世界上最有效率的工人，但是如果你的经理不知道，也将毫无意义。所以真正的问题是：如何有效地与你的经理沟通。

- 使用任务清单（见实践 10），即使只是使用个人的任务清单。这会帮助你组织自己的工作，并了解为什么这样做。如果你自己都不了解，就不可能向经理讲解清楚。如果你仍然坚持这么做，你的讲解听上去也会很没有条理，甚至很愚蠢。不过，如果有任务清单而且保证更新，就可以告诉经理此时此刻你在做什么。

  不要在真空里准备任务清单，要定期让经理审查。他们会检查你的工作计划和优先级。如果他们希望你做另一个领域的工作，就会告诉你。如果你现在的工作恰好是你该做的，就会进一步加深经理对你的好感。不论哪一种情况，他们都会对你的工作很满意，因为他们帮助你确定了方向。

- 让经理掌握你的最新进展。如果你不能至少每周与经理面对面交流，起码要发一个电子邮件，非常简练地说明你在做什么工作。不要变成一个冗长、详细的报告（除非你的经理这样要求）。要保证简短，切中要点。例如，可以说："我帮助 Trev 解决了 AIX 上的一个安装问题。"而不要说："Trev 和我工作了 4 个小时想要在 AIX 5.1 上安装 Gizmo。最后我们的安装程序在 5.2 上工作得很好，但是 5.1 在打开的文件句柄数方面还有一个问题。与 John 和 Mark 查看这个问题之后，我们最终确定这实际上是安装程序代码中的一个底层问题，我们联系了 Steve 来修正这个问题。"前一个说法相当简洁，第二种表述就过于繁琐了。如果必要，可以把状态报告分为两部分，前面是对你所做工作的一个总结，后面是各项工作的更多详细信息。

> **技巧 32**
>
> 公布你在做什么以及为什么这么做

如果老板每小时都来要一个状态更新怎么办

如果你为一个谨小慎微的经理工作，他每天来三趟查看一个为期 6 个月的项目，可以直接给他看任务清单（见实践 10）。告诉他一个小时前没有完成的那个任务现在还是没有完成。每次他们找你时，你就用任务清单作为一个可视化辅助工具。这样一来，你可以“训练”他们只是来查看任务清单，而不再不停地打断你的工作。

另一个可取的办法是鼓励你的经理主持每日例会（见实践12），并在每日例会上报告你的工作状态。如果经理能够在特定时间得到定期的状态更新，他们就不会再那么频繁地打断你的工作了（特别是你要向他们指出这些中断会对你的生产效率造成多大的危害）。

26 团队不能很好地合作

你有一组开发人员在同一个团队工作，但是他们相互之间从不交谈，从来不共进午餐，甚至从来不在公共休息室谈论头天晚上的比赛。怎么能让这群特立独行的陌生人真正开始交流，而不再躲在自己的办公室里呢？

- 如果还没有每日例会，那么现在就要开始有（见实践12），或者让你的经理着手开始安排。在每日例会上，每个人都要说话。这种交互尽管简单，尽管带有强制性，但确实能迫使个性羞涩的人畅所欲言，效果相当地好。
- 提交代码之前让团队成员相互审查代码（见实践13）。通过强制代码审查（而且轮换审查人员），就可以要求团队相互交谈，并讨论工作。他们会分享观点，相互学习技术风格和能力。如果你是一个开发人员，也可以开展一个活动，让其他团队成员来审查你的代码。心态要放开，而且要友好，不论发生了什么都不要有戒备心理。要告诉大家这样做带来了多大帮助，使你的生产效率得到了多大提高。让代码审查看上去很有吸引力，这样一来，团队的其他人也会开始要求代码审查。
- 每周搞一次午餐聚会。这种定期的非正式午餐是建立友谊的一个很好的方法。只需要看你的同事如何决定去哪家餐馆，你就能了解很多！

技巧33

会面才能建立真正的团队

大多数建立团队的活动都是孤立进行的，比如安排在周末或者举办专题讨论会。我们并不是全盘否定这些活动的价值，不过要提醒你的是，建立团队是一个持续不断的、每天都在进行的活动。如果你选择开展这些建立团队的特别活动，一定要以每天的活动作为补充。每天持续不断地建立团队才最有效。下一次团队增加一个新成员时，他不需要太多努力就能自然融入。

27 在根本问题上无法得到“认可”

你无法让团队成员、管理层或干系人参与某些关键的项目实践或过程。你已经做了宣传，明确了好处，提供了演示，甚至已经强制干预（如果你有管理职能），却仍然没有成效，没有人照你说的去做。如果是这样，就应该放弃这些人，另找愿意听的人。不过在放弃之前，先试试下面的建议。

作为经理

把新实践或过程“推销”给团队，不要只是干巴巴地宣布政策。不能一味地宣传，要实际演示。不论概念有多好，人们对一个实际示例的反应都会远远超出单纯的演讲。这意味着，首先你自己必须学会，或者选一个主动的人学习这种实践。你希望有一个内部专家，团队的其他人可以向他提出问题、要求解决难题，更一般地还可以问他“我这样做对吗？”。要选一个对新手也很耐心的人，他可以很好地沟通，能深入研究解答难题，而且不是成天扑在项目的某个关键部分上不能分身。另外团队的其他人必须尊重这个人。不要选一个忙得团团转的团队成员，因为他没有充足的时间。如果你的团队尊重率先尝试的那个人，他们也会同样地尊重这个实践。

要让你的团队能轻松改变。在一个舒服、宽松的环境中学习新东西本来就很难。如果进度很紧或者要求立即完善，那么学习新东西就更是难上加难。这样不合适，要给他们留出额外的时间来尝试这种新实践，如果需要还可以提供辅导书或培训。应当选择合适的时间引入新实践或过程（见实践 28)。（产品交付日期前三天就不是一个合适的时间！）

因特网：世界上最大的培训手册

你的预算里是不是没有足够的资金来提供培训？可以让你的团队访问因特网，并充分加以利用。现如今，不论是什么，因特网上的有关信息总是比其他任何地方都多，而且大多数都是免费的。（当然，你不能完全相信在因特网上找到的所有信息，要有保留地看待你找到的结果。）所以不要限制团队访问因特网。当然，他们可能会访问一些与工作无关的网站，但还是找到的好东西更多，所以你是很划算的。

向团队展示这些实践对他们个人有什么好处。让他们明白这样做对其自身利益的价值（换句话说，他们自己可以从中得到什么？），以此来吸引他们。如果你说“你应当这样做，因为这会改善产品质量”，对大多数人来说并没有任何激励作用。“用用这个，这样你每天 5 点就能下班回家了”，这样说就会有效得多。另外不要吝啬奖励。设定一个目标，在团队达到目标后提供一个奖励。可以小到一盒多纳圈，也可以大到所有相关人员得到一大笔奖金。不过一定要为团队定一个具体的目标。

最后，如果这个团队确实不愿意前进，就只好换一些希望前进的人了。尽可能多地为团队提供机会来采用这个新实践。如果他们坚持不采用，而且不采用又根本无法完成任务，就只能放弃他们，另找更合适的人。如果你坚持用这些人，项目就会失败，那时你也将被辞退而别无选择。虽然话不中听，但是要记住：如果一个船长翻过船，将很难再受命上船……

作为开发人员

如果你是一个团队成员，前面对经理所说的大部分内容同样适用。尽管你没有经理的权力，但是在这种情况下并没有太大影响。你要推销这种实践，而不只是宣传。要亲自使用，成为一个专家，然后向团队的其他人展示它对你的帮助有多大。寻找机会利用这个实践帮助同事解除困境，然后着手提供帮助。（当然你也可以 5 点就下班回家，不过这样一来以后你的团队就可能不太会想到你了。）同样地，如果你的团队拒绝学习一个能保住项目的实践，就只好投奔其他团队了，不过至少你已经尝试过。

作为客户

首先，要认识到对于开发团队的内部实践你没有太多发言权（而且你也不希望有）。只要团队能交付一个好产品，具体如何工作对你来说并不重要。你感兴趣的实践应该能够让你尽早而且尽快地发现和解决项目中的问题。

其次，沟通和反馈是关键。尽早而且经常与团队沟通，给他们提示，对从他们那里得到的所有东西给出详细的反馈。当然，这里假设你确实得到了一些东西。项目团队必须能经常提供演示系统，你可以亲自尝试使用。演示系统比文档好得多：演示系统是客观真实的，文档只是表面文字。如果演示系统还不能满足你的期望（或者你根本没有得到演示系统），就必须更积极地参与项目。

怎么让开发团队改变工作方式呢？下面给出一些建议。

- 详细记录演示系统中还缺少什么，然后确认下一个演示版本确实修正了这些问题。如果还没有，可以要求团队做出补充说明，描述将如何应对你发现的问题。
- 请一位专家与团队一同工作。找一个既能向团队传授更好的实践，还能监督他们的进展的人。
- 如果情况很糟糕，那么再与联络人协商，要求他们使用更好的实践，然后密切监督他们的进展。不过这是没有其他办法时的最后一招。

最后，如果很明显这个团队的做法根本无法交付你想要的产品，而且他们不打算为此做必要的工作，你就得另找其他愿意做这些工作的团队了。不要白白扔钱坐等团队奇迹般地改进。应当减少你的损失，另找一个更好的团队。

28 新实践没有帮助

这本书里我们一直在说“站在巨人的肩上”。要做到这一点，最好的办法就是学习巨人们正在使用的实践，并亲自着手应用。所有软件工作室都有改进的空间，所以一定要注意有没有更好的实践可以提供这种改进。

什么时候不要引入新实践

先讨论这个问题看上去有些奇怪，是不是？不过很多工作室确实会在不必要或者不恰当的时间增加新的实践和过程（这会对关键工作造成破坏）。最后这些新实践导致的问题比解决的问题还要多。在考虑是否应当增加实践时，首先要问自己：“现在是恰当的时机吗？”

如果没有需要修正的问题，就不要尝试引入一个新实践或新过程。绝对不要因为一个实践是“正确做法”就盲目引入。相反，要找出你的工作室真正的问题，然后明确如何修正。这样一来你会拥有一个平稳运转的工作室，因为你只修正有问题的地方。

技巧 34

只修正需要修正的地方

考虑问题和修正时要有创造性。“条条大路通罗马”[①]并不只适用于 Perl 程序！成功管理一个软件工作室有很多方法。没有哪组实践或哪个过程适用于每一个项目的每一个团队。

引入一个新实践之前，还需要考虑工作室还有哪些其他工作。一个主要版本发布前 3 天可能就不适合突然改变你目前使用的过程。正当团队昼夜奋战努力推出产品时，他们很难接受任何让他们分心的事情。一个较好的时间可能是产品交付后的一周左右，这样团队在适应这个变化之前还可以有一个喘息的机会。应当选择一个合适的时间引入实践，尽量减少对关键活动的破坏。

技巧 35

破坏性的“最佳实践”不是最佳实践

① Perl 的格言，写在 Perl 手册页面最下面。

当然，要确保你考虑采用的实践或过程确实能带来改进。如果它不能加快运行速度，提高运行效率，你的团队就不要也不应该采用这个实践。在团队看来，“如果这个实践能让我 5 点就下班回家，那我就接受”。

如何引入一个新实践

一旦决定增加一个新实践作为你的常备技能，如何有效地做到？要做到两点：演示和说服。完成这两个目标后，你马上就会得到一群狂热的粉丝。

你必须获得多个不同人群的认可。首先也是最主要的，是那些将具体采用这个新实践的人，也就是你的团队。如果他们对这个新实践兴趣不大，别人再有兴趣也没有用。你是不是听过很多这样的故事：一些公司迫于上层压力不得不采用一些新方法，这些强制行为有多少确实让工作有了改进？（答案：并不多。可以想想全面质量管理、ISO-9000，等等。）

相反，你是不是经常听说一些“草根”技术（底层引入的技术和实践）像野火般席卷整个团队、整个公司甚至整个行业？（可以想想 UNIX 与大型机、PC 与 UNIX、敏捷开发与瀑布软件开发模型。）

技巧 36

自下而上改革

那么怎么让你的团队热衷于这个新想法呢？还记得我们之前讨论的演示吗？要向他们展示这个过程或工具，不要只是说说而已。具体来讲，就是要向他们展示这个过程或工具出色的工作表现，特别是与老办法做比较。

如果你认识一个“狂热的粉丝”，他确实使用了这个实践并取得了很好的效果，可以请他来，让他向你的团队展示这个实践带给他的帮助。更好的做法是你亲自使用。你的生产力和效率得以提高将成为最有力的证据，然后告诉你的团队。也许你根本不用多说，因为团队可能已经注意到了你的改进！关键是要证明这个全新的想法确实名不虚传。

技巧 37

要具体展示，不要光说不练

说服你的团队使用这个新实践至关重要。不过，并不是说你可以完全忽略

管理层。如果经理能认可，实践的引入会更为容易。你的经理可以说服那些对这个想法有异议的团队成员做个尝试，而且可以作为屏障挡住来自公司其他方面的阻力。有了这个屏障，你的团队就有时间找出实现这个实践的最佳方法。显然，如果工作时不用躲躲藏藏，团队将更容易做出改变。

如果无法得到管理层的认可，也不要因此妨碍你采用这个实践，特别是如果团队有很高的热情时。可以把它当作一个“隐形实践”，充分使用但悄悄地进行。类似代码审查等实践不需要管理层认可也能有效完成。你可以使用这些实践，团队以外的人甚至不知道你正在做代码审查。你可以在自己的机器上建立一个 CI 系统，并与你的团队共享结果。取得一些关于这个实践的经验之后，你就可以向管理层展示这个实践，并指出它的相关好处。

技巧 38

让管理层逐渐认可

29 没有自动测试

在一个有自动测试套件的工作室工作过之后，你肯定不愿意再退回到没有自动测试套件的工作室了。不过如果你所在的工作室确实没有自动测试该怎么办呢？采用什么方法才能最容易地为工作室引入自动测试？

对于自动测试，人们最不满的就是维护太过麻烦。如果自动测试运行不频繁，运行间隔期往往会出问题。运行之间间隔的时间越长，测试就越有可能失败。如果测试失败后太长时间不管不问，再想修正它就会变成一个维护恶梦。

所以先来解决这个问题。开始编写和提交测试之前，一定先要有一个 CI 系统（见实践 4）。建立一个 CI 系统，使它每次代码变更时都运行测试。

如果每次有人提交代码时都会运行测试，而且这个人会得到通知，那么一旦出问题就可以修正测试。像这样每次修正一两个测试比产品交付前修正一大堆测试要容易得多！

如果已经有一些可以手动运行的测试代码，就要将这些测试移植到 CI 系统中。这样一来，系统中很快就会有更多测试（除非测试代码确实像一团乱麻）。先试着移植几个测试来看看是否容易。如果移植看起来相当麻烦，得不偿失，就不如干脆放弃努力，另外创建一个新的测试套件。

接下来，使用模拟客户测试方法编写你自己的测试，这样各个测试可以得到最大回报。这时已经没有时间为代码中的每一个方法编写一个单元测试了。模拟客户测试更为高效，因为一次就可以完成大量代码的测试。

使用缺陷驱动测试来明确要编写哪些测试。这样可以在最需要的地方增加测试，相应地可以发挥最大功效。现在，只有当一段代码出现一个活跃的 bug 时才为它增加一个测试。这意味着所增加的测试将解决代码中当前存在的问题，这样就可以尽可能从每个测试中得到最大好处。

技巧 39

测试有 bug 的代码

30 我们只是低级别开发人员，没有人指导我们

你是工作室里为数不多的高级开发人员之一，工作室里还有大量低级别或中级开发人员。作为师傅你怎么把自己的经验传授给这些学徒，还要保证自己不因工作量太大而过度劳累？

首先，请你的团队领导人或经理召开团队的每日例会（见实践12）。让每日例会成为一个没有压力的论坛，在这里低级别成员可以任意讨论他们的问题，而不必明确地请求帮助。然后你可以在这里分享解决方法，这样就不必单独地给每一个团队成员面授机宜了。

其次，引入代码审查（见实践13），确保你或者另外某个高级开发人员参加每一个审查。代码审查非常适合进行一对一辅导和交互，所以除了能够确保代码正确地运行外，它还有很多作用。你可以针对更广泛的问题指导低级别开发人员，如讲解算法效率和编码风格等等。

一段时间之后，低级别成员就会从前人那里学到好习惯，也开始像高级程序员那样谨慎行事。

31 我们在一个“死亡之旅”项目中

如果你连续工作几小时甚至好几周（通常是因为管理层强加的不合理的最后期限），这就说明你正在一个“死亡之旅”项目中。平时每天工作 10 到 12 个小时，周末还要加班，这已经成为常事。不过你这样做是为了项目好，对不对？这正是成功与失败的差别！不过……

首先，要认识到“死亡之旅”并不是一种写软件的好方法。你精神上会很疲惫，长时间工作也很容易让你犯错误。如果一天工作 12 个小时，你可能会在第 10 个小时“铤而走险”地走捷径。如果你一周工作了 80 个小时，可是产品还是达不到标准，那你可真不幸。

很多书会告诉你如何熬过“死亡之旅”[You99]，不过我们来看如何克服。

首先，建立一个新的项目进度。使用任务清单（见实践10），对每个任务设置估计时间。确保这些是比较实在的时间估计，而不是“死亡之旅”式的估计。与技术领导人一同确定正确的优先级，并与管理层达成一致。

其次，根据任务清单上的时间估计，为项目建立一个时间表。（不论是你个人的任务清单还是整个团队的任务清单，这一点都适用。）公开这个进度。把它写在白板上，或者公布到网站上。通常制订进度的人创建时间表时并不了解涉及哪些工作。你要帮助他们了解。

你的新进度可能会超过现在的最后期限，这没关系。你想要的是准确了解一个给定日期项目的真实状态，而不是虚构一个时间表来支持一个虚假的大结局。

关于最后期限

“我喜欢最后期限。我尤其喜欢它们飞驰而过时发出的嗖嗖声。”——Douglas Adams

一旦对实际能完成多少工作有了更准确的认识，你就可以把这个时间表展示给经理看。告诉他们你认为这个项目有麻烦。管理层可能不乐意听到你的预测，不过我们的目标并不是图眼前高兴。试着告诉他们如果进度与现实

不符，肯定将无法实现（不过有些情况下，可能出于其他原因还是决定保留这个不合适的进度）。

现在你有两个选择：推迟日期或者删除特性。（或者辞职，这可以算是第三个选择。）团队会做出怎样的选择取决于哪个更重要，是交付日期还是特性。如果你决定推迟日期，要保证有一个活动的任务清单，使得在不调整时间表的条件下，任何人都不能再增加新特性。

现在你已经展示了你的组织能力（希望如此），并做出了一个合理的计划。如果你的计划是准确的，你可能会被重用，以避免下一个项目遇到同样的问题。

32 特性不断蔓延

除非你特别擅长预测未来，否则你最初制订的任务清单不会让每一个人都满意太久。你会得到一些建议，要求增加新特性，声称这会让你的产品“大获成功”。你如何决定是否增加这些特性？另外，如果你拒绝了一些人的建议，又如何说服他们你的决定是正确的？

有人请求一个新特性时，要查看团队当前正在实现的任务。下一次交付之前有没有时间来实现这个新特性？这个新特性与已经列入计划的特性是否兼容？另外对计划交付的产品来说，这个特性有意义吗？

如果对这些问题的回答都是肯定的，就确实应该把这个特性增加到任务清单中去。确定它与其他特性相比有多重要，为它指定一个优先级，并指派一个开发人员来实现它。然后你可以放心地睡大觉了，因为你已经在任务清单中为这个特性指定了合适的位置。

不过，如果对以上问题的回答有一个是否定的，则不论请求者给你多大的压力，都不要增加这个特性。你要柔和但坚定地利用任务清单来解释你为什么不去实现这个特性。如果请求者是一个明理的人，能接受合理的解释，这应该不难。如果请求者根本不容反对，你就礼貌地找一个借口继续回去工作。对这个问题继续讨论纯粹是浪费时间。

在一个小公司，与我们一同工作的人经常对产品提出新特性。有时这些特性是有道理的，但是通常都属于“无用”特性，或者不算太重要，不值得重新指派开发人员来实现那些特性。这是一个小型初创公司，资金很紧张。交付推迟就意味着公司没有足够的钱来发工资。我们把任务清单写在开发人员办公室的白板上（没错，公司太小了，所有开发人员都在一间办公室里!），让大家都能看到，并为任务指定了优先级。经过简短讨论，大家通常就会很清楚为什么我们不去实现那个声称“今天就必须实现的特性”了。

33 我们永远也完不了

假设你的公司销售一个复杂的软件产品，而你是开发下一个版本的团队领导人。你将如何决定要做些什么？可以使用任务清单将产品划分为许多单个特性（见实践 10）。如果任务清单上的所有特性都已经完成，而且能很好地集成（当然，特性集成也应该是任务清单上的任务之一！），就说明你的项目已经完成。

我们曾经到过这样一家公司，他们开发一个产品已经快两年了。这个产品永远都是演示模式，因为没有人知道如何完成第一个版本。干系人总是时不时提出不同（而且经常是相互冲突）的需求，往往只是因为近期与一个潜在客户有过交谈就产生了新的想法。代码中只有一堆完成了一半的特性"纠缠"在一起，只能在演示时简单展示。没有哪个特性完善到可以认为已经完成，而且有很多特性尽管大家都认为没有意义，但还没有从代码存储库中删除。我们首先做的是创建产品包含的特性列表，为产品建立一个"全局图"。这样一来，我们就得出了哪些特性要在产品的第一个版本中完成，而且这个版本终于在几个月之后交付了。

以下是关于特性列表的一些指导原则。

- 如果任务的估计时间超过一周，就要把它分解为子任务。一个顶层任务需要几周或几个月完成是可以的，但是这个估计只是一个猜测，除非你用子任务的时间估计来支持它。
- 不到一天就能完成的任务粒度太小。如果一个任务可以在不到一天的时间内完成，说明它可能过于底层，不应当放在任务清单中。
- 一个客户示例（用例或场景）可能涉及任务清单中的多个特性。不要试图把整个示例都放在任务清单的一个底层任务中，应当把它分解为子任务。
- 为任务清单中的任务指定优先级，并坚持按这个优先级行事。不要在第一优先级任务未完成的情况下去处理第二优先级的任务。不过在必要的情况下也可以改变这些优先级。
- 指派特定的人来完成任务清单上的各个特性。可以动态分配（某个人完成一个任务时，会"分配"到另一个任务），也可以提前分配好，然后根据需要调整。这取决于你的团队最适合哪种方式。
- 要灵活。充分利用变更。改变任务清单意味着你在不断改进和求精，或

者得到了客户的反馈，或者在努力保证任务清单与客户的实际需要一致。

这就像一架飞机停飞的场景。机场的乘客听说后都很不高兴，只有一个人除外。他快活的解释出乎所有人的意料。他说："大家想想看，如果飞机停飞，肯定是飞行员、飞机或者天气中某个环节出了问题。不管怎样，我宁可呆在地上，也不愿意在有问题的情况下飞上天！"你也可以这样来考虑任务清单。如果你改变了特性，这是因为原来它们有问题。实际上你原来是在构建错误的产品。尽管这个修改现在看来很让人恼火，但是如果等到完成产品之后才发现必须把它全盘丢掉再重新开始，那显然更糟糕。

技巧 40

任务清单是一个活动的文档，生活中处处有变化

作为开发人员

即使你的团队没有使用任务清单来跟踪特性，你也可以自己使用。把任务清单写在你的白板上，列出要完成的所有任务，然后让技术领导人帮助你设定优先级。如果他们不帮忙，那你就自己动手设置，但是不要硬性规定哪个任务更重要。这是技术领导人的工作。现在你的工作就是可见的、透明的、可以审计的了。

作为经理

让任务清单成为项目管理层的核心工具。直接按照任务清单来设置团队的工作安排。任务清单可以包含顶层任务、底层任务，或者这二者的混合。一定要灵活。要保证任务清单已经指定优先级。否则，团队必须猜测哪些任务最重要，而且从一般经验来看，他们往往会先选择最酷或者最容易的特性，而不是必要的特性（恰好也是最难、最麻烦的特性）。

完成一个任务之后不要简单地把它从任务清单上去除。要留下完整特性的一个副本（包括日期、优先级和指派的开发人员），这可以作为你的审计记录。如果有人问你"为什么不增加特性 X"或"增加了哪些特性"时，你就能从这个任务清单中找到答案。

技巧 41

如果任务清单上没有，那就不是项目的一部分

产品交付之后，不要忘记再来查看任务清单上延误的目标日期。检查进度的延误会提高你将来的预测能力。不要因为“产品已经交付”就止步。要注重细节但是不要吹毛求疵，你的目的是找出延误的原因从而可以修正。毕竟，不重视历史的人肯定会重蹈覆辙。吸取这些教训，否则你就会重复以前的失败。

作为客户

向开发团队要任务清单的一个副本。不要只满足于顶层任务。你不必花几天通读整个任务清单，不过你肯定想知道开发团队确实已经详细掌握了工作范围。如果团队延误了一些里程碑，就要与开发经理坐下来谈一谈，查看任务清单，并找出原因。这会让你清楚地了解开发团队当前到底在做些什么，并帮助他们更有条理。（如果他们没有使用任务清单，这会要求他们现在就开始使用！）

如果任务清单上的特性优先级与你的优先级不一致，要尽快反馈。如果你没有说出你要什么，就一定得不到你要的东西。你的等待只会浪费一天又一天的开发时间。要记住，一个团队处理一个特性集的时间越久，改变特性集时他们就会越恼火。所以一拿到任务清单你就要马上审查，立即发出反馈（包括正面和负面反馈）。及时的反馈会有惊人的促进作用。不要浪费这个机会。

技巧 42

要快速做出反馈

附录 A

技巧汇总

1. 选择习惯
2. 留在沙箱里
3. 如果需要就将其签入
4. 从第一天起就使用脚本构建
5. 任何机器都可以作为构建机
6. 持续构建
7. 持续测试
8. 避免集体失忆
9. 演练产品——自动测试
10. 使用通用、灵活的自动化测试框架
11. 工欲善其事，必先利其器
12. 使用开放格式集成工具
13. 使用熟悉的关键路径技术
14. 按照任务清单工作
15. 要有一个技术领导人
16. 通过每日例会频繁进行航向修正
17. 可以说“以后再来”
18. 经常审查所有代码
19. 目标是软件，而不是遵从过程
20. 集体参与建立架构
21. 如果生产环境会用到，你也要用到
22. 先解决最难的问题
23. 封装的架构是一个可伸缩的架构
24. 除非船在移动，否则转动方向盘没有用

25. 测试之前不要修改遗留代码
26. 使用测试驱动重构清理不可测试的代码
27. 模拟客户测试可以事半功倍
28. 持续测试不断改变的代码
29. 必须适用于所有人
30. 经常集成，并持续构建和测试
31. 尽早而且经常发布真实演示系统
32. 公布你在做什么以及为什么这么做
33. 会面才能建立真正的团队
34. 只修正需要修正的地方
35. 破坏性的“最佳实践”不是最佳实践
36. 自下而上改革
37. 要具体展示，不要光说不练
38. 让管理层逐渐认可
39. 测试有 bug 的代码
40. 任务清单是一个活动的文档，生活中处处有变化
41. 如果任务清单上没有，那就不是项目的一部分
42. 要快速做出反馈

附录 B

源代码管理

源代码管理（SCM）程序，通常也称为版本控制系统，可以跟踪你的代码和对代码的修改。另外，好的 SCM 可以将特定版本的代码与重要的里程碑（如产品发布）关联起来。

可用的 SCM 系统

CVS .... http://www.cvshome.org

这是一个免费的开源客户–服务器SCM，可以完成你需要的所有工作，只是命令行界面有点令人费解。全世界大大小小的公司都在使用CVS。

Subversion .... http://subversion.tigris.org

自称是 CVS 的替代产品，可以完成 CVS 的大部分工作，另外还增加了大量其他功能。类似于 CVS，它也是开源而且免费的。

MS Visual SourceSafe........ http://msdn.microsoft.com/vstudio/previous/ssafe

这是微软的 SCM，微软的很多开发工具和 IDE 中都集成了这个工具。如果你的工作室购买了微软的开发工具，那你可能已经为 SourceSafe 付过钱了，已经得到了一个可以使用的许可。

BitKeeper... http://www.bitkeeper.com

BitKeeper 是一个商业产品，Linux 内核开发人员使用这个工具已经很多年了，而且他们报告生产力确实有了显著提高。

关键概念

存储库（repository） 存储源代码的位置。

工作空间（workspace） 你的机器上的源代码的本地副本。代码要从存储库签出到你的工作空间，经过一些处理后，再把它重新签入存储库。

客户程序（client） 这是你的机器上运行的程序，它通过服务器与存储库交互。

服务器（server） 位于存储库前面的程序，可以处理客户程序。

分支（branch） 为项目建立分支，从而可以有多个开发路径。例如，一个项目分支完成当前版本的 bug 修正，另一个分支用来开发下一个版本。

标签（tag） 可以采用这种方法来标识一个特定版本的文件、目录、项目等等。

合并（merging） 两个或多个开发人员处理同一个文件时，必须合并变更。

锁定（locking） SCM 采用这种方式来确定谁可以对一个文件做出变更。在一个悲观加锁系统中，一次只有一个人可以对一个文件做出变更。而在一个乐观加锁系统中，可以有多个人做出变更，文件签入时会自动合并所有变更。

如何选择

> **工具成本**
>
> 很多开发团队喜欢使用免费的工具，还有一些团队则倾向于有产品服务支持的商业产品。我们更喜欢那些可以免费得到而且可以从用户群体得到支持的工具，不过我们不会根据成本来选择工具，而要根据哪个工具适用来做出选择。如果有多个工具都能满足我们的需要，就会选择其中的免费工具。你要知道哪些工具适用于你的环境。

特性

标签——为特定版本的源代码建立标签是否容易？使用标签来访问这个代码是否容易？

合并——是手动合并还是自动合并？

多项目项目（multiproject project）——项目中可以包含其他项目、模块和版本吗？定义项目之间的依赖关系是否困难？

易用性

你的团队使用的编辑器或 IDE 是否集成了这个工具？可以嵌入到你的构建脚本中吗？要记住，如果使用不方便，那么谁都不会使用。

可伸缩性

系统能否处理你的所有文件、项目和用户，而且不会丢失或破坏文件？

性能

基本操作是否足够快而值得人们去做？或者人们会不会因为不想等待而绕过这些操作？

更多有关信息

见实践 2。

附录C

脚本构建工具

脚本语言可以帮助你完成构建产品过程中的大量工作。基于编译和组装产品的有关特性，利用脚本语言可以掌握构建过程的每一个步骤，从而做到可重复而且易于自动化。

可用的脚本工具

操作系统脚本语言（如shell、批文件）

这是现有操作系统提供的一些脚本语言，不过往往过于一般，缺少你需要的很多常用功能。使用这些工具，你很可能需要一切从头开始。

make

make .... http://sources.redhat.com/cygwin

所有现代 Unix 和类 Unix 系统上都提供有 make，还可以从以上 URL 得到面向 Windows 的一个免费版本。

make 堪称所有构建脚本的鼻祖，已经有数十年历史。make 之类的工具可以对你稍有促进，不过仍然要求你自己编写代码来完成很多常用功能。正是从 make 开始引入了跨平台脚本的思想。

Automake .... http://www.gnu.org/software/automake

Automake 是一个 Perl 工具，可以帮助你创建 make 文件。

语言特定的工具

很多工具是针对一些特定语言编写的。例如，Ant 就是为 Java 语言创建的，可以创建 JAR 和 WAR 文件，生成 JavaDocs 等等。使用类似 Ant 的工具可以帮助你避开创建 WAR 文件的细节，同时还能提供一种健壮而且可重复的方式构

建 WAR 文件。

Ant .... http://ant.apache.org

Ant 是一种面向 Java 的标准构建脚本语言。它提供了大量内置功能，非常灵活，完全可以面向更多语言建立脚本，而不只是 Java。

NAnt .... http://nant.sourceforge.net

NAnt 是.Net 版本的 Ant。

Groovy .... http://groovy.codehaus.org

尽管 Groovy 确实是一种通用脚本语言，但它还允许从 Java 代码访问 Ant 的所有功能。其目的是提供 Ant 目标的方便性，同时还能提供真正编程语言的强大功能。Groovy 仍在完善，所以使用要慎重。

Rake .... http://rake.rubyforge.org/

Rake 是一个面向 Ruby 的构建工具，与 make 的功能相似，不过它使用纯 Ruby 作为脚本语言。还支持规则模式和有前置条件的任务等特性。

通用脚本语言

严格地讲，脚本语言并非针对构建系统，但是由于很多人已经了解了某种脚本语言，要求创建一个构建系统时，通常就会让通用脚本语言“赶鸭子上架”。坦率地讲，我们从来没有见过一个好的构建系统是用通用语言建立的。增加这一节主要是为了完整性，但是强烈建议你使用一种专门为这个任务而设计的工具。

Ruby .... http://www.ruby-lang.org/

Ruby 是一种日益流行的脚本语言。它有很多内置的面向对象特性，可以像 Perl 一样处理文本文件。Ruby 很容易使用，非常简洁，而且拥有活跃的用户群体。

Python .... http://www.python.org/

Python 也是一个包含很多面向对象特性的解释语言。对 Python 的格式有人喜欢有人反感，不过那些铁杆粉丝对这种语言的热情令人难忘。

Perl .... http://www.perl.org/

如果不提到 Perl，就不能算完整地讨论了脚本语言。Perl 已经存在多年，每个主要平台上都有提供，计算机能做的工作几乎都可以用 Perl 完成。Perl

有一组极其丰富的功能来处理文本文件，另外“Web 归档网络”[①]包含了大量 Perl 代码，可以实现你要完成的所有工作。它的语法很晦涩，不过功能是有目共睹的。

构建系统

Maven .... http://maven.apache.org

Maven 之类的工具可以让你更上一个层次。对于这类工具，主要的不满是它们封装得过头了。Maven 对于构建位置和任务名有特定要求，不过你可以绕过这些限制。大多数人对这种工具非爱即恨。你可以自己试试看！

Maven 2 .... http://maven.apache.org/maven2/index.html

Maven 的下一版本，这个版本得到完全重写。构建模式比 Maven 1 中更简单，性能也得到了显著提高。

关键概念

语法（syntax）　工具使用的语言，make 之类的语法很晦涩，不过大多数延迟模型脚本都是基于 XML 的。

任务（task）　工具能实际完成的工作。至少需要编译和链接特性，但是像 Ant 之类的工具可以完成更多任务。

脚本解释器（script interpreter）　执行脚本的“引擎”。

如何选择

全面性

是否无需编写大量定制代码就能做到所需的一切？

可读性

工作室里的任何人都能阅读并理解脚本吗？

平台可用性

是不是所有平台上都可用？

可伸缩性

速度是否足够快？可以处理你的工作室所需的工作负载吗？能处理一两个

① http://cpan.perl.org。

项目与处理 50 个项目是有很大差别的。

可扩展性

增加所需的额外功能难度有多大？

灵活性

能采用你希望的方式使用这个工具吗？或者这个工具是否要求你改变原来的工作方式？

与编程风格的一致性

可用并不意味着对你最适合！

可以容易地完成构建任务

你能在 30 分钟内学会吗？还是需要两周时间才能学会？

更多有关信息

见实践 3。

附录 D

持续集成系统

每次对项目做一个变更时，CI 系统会自动构建项目。它会监视一组资源（如源代码存储库、文件系统，甚至另一个项目），发生变更时，CI 系统就会启动构建脚本。构建完成时，CI 系统会把构建结果告诉你（或任何其他重要人物）。

可用的系统

CruiseControl .... http://cruisecontrol.sourceforge.net

这是一个开源的 CI 系统，用 Java 编写，CruiseControl 提供了大量功能，还有一个活跃的开发人员群体。我们使用的就是这个 CI 系统。

CruiseControl.NET .... http://sourceforge.net/projects/ccnet

面向.NET Framework 的 CruiseControl。与面向 Java 的 CruiseControl 特性集稍有不同，但是概念是一样的。

DamageControl .... http://damagecontrol.codehaus.org

用 Ruby 编写的一个 CI 系统。它的设计目标是快速而简洁，这个系统提供了一个很好的 Web 界面，可以与很多其他工具内置集成。

AntHill .... http://www.urbancode.com/projects/anthill

AntHill 被称为一个构建管理服务器，要求构建过程采用它自己的构建机制。不过，AntHill 有一个很好的 Web 界面，而不是使用配置文件，这使得构建项目几乎轻而易举。AntHill 提供了开源和商业版本，不过只有商业版才提供额外的特性。

Continuum .... http://maven.apache.org/continuum

Continuum 是一个新的 CI 系统，设计目标是与 Maven 紧密集成。它有一个

“零配置”特性，可以与一个现有的 Maven 项目无缝集成。

关键概念

配置（configuration） 如何建立系统。很多 CI 系统使用配置文件，但是有些系统（如 AntHill）使用一个基于 Web 的界面。

CI 引擎（CI engine） 具体监视变更并执行构建。

外部接口（如 JMX、RMI、Web 页面、COM、XML-RPC） CI 系统运行时进行控制的方式。

支持的构建工具 CI 系统能完成的工作（如编译、链接、部署、安装）。

与其他工具集成 从 CI 系统运行新工具。

通知机制 向 CI 系统用户通知一个构建是否成功的方式，可以包括电子邮件、Web 页面、RSS 提要或熔岩灯。

日志和测量 跟踪和表示构建通过（或失败）时已构建部分的有关信息，以及构建是否通过。

如何选择

可以容易地构建项目

CI 系统能不能完成项目构建而不必大费周章地进行配置？

在你的平台上运行

能不能在你需要的任何地方运行？

与其他工具合作

能不能很容易地结合代码性能分析工具、安装工具、部署工具等等？

扩展到需要构建的项目数

如果要构建大量项目，就需要一个速度很快的 CI 系统。

完全自动化

不应需要任何手动步骤。

提供适当的通知

能不能以一种可见的方式告诉你发生了什么？

更多有关信息

见实践 4。

网上还有一个非常棒的特性矩阵[①]。

① http://docs.codehaus.org/display/DAMAGECONTROL/Continuous+Integration+Server+Feature+Matrix。

附录 E

问题跟踪软件

问题跟踪软件会管理项目的 bug 列表、要完成的工作以及其他重要的任务。如果得到合理使用，这些列表将会成为项目的“记忆”。

可用的软件

Bugzilla .... http://www.bugzilla.org

这是一个基于 Web 的开源 bug 跟踪系统。Bugzilla 得到众多组织的广泛使用，包括 Mozilla 和 Red Hat。

JIRA.......................................http://www.atlassian.com/software/jira/default.jsp

这是一个“购买之前先试用”系统，为电子邮件、RSS、Excel、XML 和 CVS 等提供了扩展。CruiseControl 项目使用的就是这个系统。

FogBugz ....http://www.fogcreek.com/FogBugz

这是 Fog Creek Software（Joel Spolsky 的公司）的另一个商业产品，Joel 正是著名的 *Joel on Software* 的作者。

PR-Tracker .... http://www.prtracker.com

这是一个企业级的基于 Web 的 bug 跟踪系统。按用户授予许可，而不是按计算机。

关键概念

问题输入（issue entry） 如何将问题的详细信息输入到系统中。有些工具使用Web界面来输入问题描述，还有一些工具要求为系统加载客户软件。

问题描述（issue description） 问题是什么，如果合适还要说明如何再生这个问题。

分配（assignment） 谁将处理这个问题。

优先级（prioritization） 这个问题相对于系统中其他问题有多重要。

搜索（search） 根据一些条件（如产品、用户、优先级等）查找相关的问题。

报表（reporting） 根据相关条件生成问题报表。

通知（notification） 系统中出现变更（新问题、问题状态改变等）时，要通知有关各方。

与其他工具的集成 与通知系统、源代码管理系统、报表生成工具等的集成。

用户特定的操作（user-specific operation） 根据用户许可来限制操作，允许客户对 bug 报告归档。

如何选择

界面

你经常做的操作是否很容易完成？能否从所需的任何地方访问系统？

- 基于 Web——只需要一个 Web 浏览器。
- 客户–服务器——需要在你的机器上安装客户端来访问系统。
- 本地——只能从一个机器访问系统。

可伸缩性

是否能处理所有项目及其所有问题？能不能应对所有用户？

不要太过复杂

如果不容易使用，谁都不会用。

跟踪你需要的信息

能不能捕获你关心的所有信息？是否会获取你不需要的信息？

生成你需要的报表

能不能提供你需要的直接可用的报表？或者是否需要对标准报表做大量定制处理？能不能提供你关心的信息？

通知选择

如电子邮件、RSS、Web 页面、Wiki 等。

对现有工具的支持或集成

如果不提供支持（集成），你就必须手动地来回移动信息。

在你的平台上可运行

如果开发人员在多个平台上工作（如 PC、Mac、Unix 等），应当确保他们都能访问这个工具。

更多有关信息

见实践 5 和实践 6。

附录 F

开发方法

有时看起来有多少开发人员就有多少不同的软件开发方法。新的开发方法不断涌现。要跟随行业的思想领袖与时俱进，看看他们又提出了什么思想，并在你自己的工作室大胆尝试。

有一种方法要避免，那就是臭名昭著的瀑布方法。在目光较长远的开发圈子里，这种方法的声誉普遍很糟糕，不过让人奇怪的是居然还有那么多工作室在使用这个方法。瀑布模型认为你能全面了解项目的每一个阶段，并为各个阶段设定一个具体的进度。瀑布方法还希望你在启动项目之前就设定好进度！那些对技术一窍不通只崇尚进度的经理一直以来都是瀑布方法的狂热粉丝。

可用的方法

曳光弹开发 .... www.PragmaticProgrammer.com

这正是我们使用的方法，我们确实很偏爱这种方法！见第 4 章。

敏捷开发 ....http://www.agilealliance.com/home

与其说这是一种特定方法，不如说它是一场运动，强调适应性、沟通和迭代。

能力成熟度模型：（CMM、SW-CMM 或 CMMI）

.... http://www.sei.cmu.edu/cmmi

软件开发过程中大多数方面的形式化模型。

极限编程（XP） ....http://www.extremeprogramming.org

这是 Kent Beck 创造的一种方法，XP 提出的规则和实践包括结对编程、迭代和大量客户交互。

Rational 统一过程（RUP）

.... http://www-136.ibm.com/developerworks/rational/products/rup

这是一个重要的形式化方法，包括很多不同方面。

Scrum .... http://www.controlchaos.com
这是一种敏捷方法，围绕增量交付周期（称为 sprint）展开。

Crystal .... http://alistair.cockburn.us/crystal/crystal.html
Crystal 是一个高适应性软件过程。它基于这样一种想法：每个项目都不同，所以每个项目都需要一种不同的方法。

关键概念

阶段（phase） 项目过程中采取的步骤系列。不同的方法有不同的阶段，不过大多都包括需求收集、编码、测试和编写文档等阶段。

里程碑（milestone） 开发过程中发生的特定事件或交付的产品。

可交付产品（deliverable） 可以交给有关方面的软件部分（例如，代码、文档、演示系统）。

进度（schedule） 里程碑必须何时完成。

工作分配（work assignment） 为团队成员分配特定任务，并跟踪这些任务的完成情况。

沟通（communication） 团队中任何人都不能在真空中工作。

如何选择

产品类型

你在开发什么软件？小项目与大项目需要不同的方法。

团队规模

与大团队相比，小团队采用一种不太正式的方法也能成功。

团队人员的类型

有些人在不太正式、监督和规划较少的环境下能更好地工作，而有些人则不然。

客户类型

客户在项目期间可能能够提供咨询，也可能并不提供。

外部约束

有时可能要求你必须使用某种特定的方法（例如，合同要求或政府审计要求等等）。

过往记录

如果当前的方法没有问题，就不要去修正！

简单性

大型的复杂方法可能不是一个小团队能够应付的。

更多有关信息

见第 4 章。

Martin Fowler 有一篇关于开发方法的文章（见 martinfowler.com/articles/new-Methodology.html），对这个主题做了很好的讨论。

附录 G

测试框架

利用测试框架，你可以组织和运行你的测试，而不用担心底层细节。在这里我们将讨论两种测试框架：自动化测试框架和测试工具。自动化测试框架（test harness）是一个 API，可以作为编写你自己的测试的基础。测试工具（test tool）是一个程序，可以用来创建和运行测试。这两种测试框架各有其一席之地，如何选择取决于你要测试什么。

可用的测试框架（自动化测试框架）

SUnit.. http://sunit.sourceforge.net

SmallTalk Unit 是原来的 XUnit 自动化测试框架。当前所有单元测试框架都由此复制。

JUnit ....http://www.junit.org

较流行的测试框架之一，用 Java 编写，并面向 Java。

JUnitPerf............................ http://www.clarkware.com/software/JUnitPerf.html

这是一组很好的JUnit扩展，可以较容易地实现性能和可伸缩性测量。这可以作为一个很好的例子，来解释为什么要使用开放工具。很多人会扩展开放工具，但如果你使用自己编写的工具包，就必须自己编写扩展包。

NUnit ....http://www.nunit.org

这是一个面向.NET 的自动化单元测试框架，最早从 JUnit 移植而来。它支持所有.NET 语言。

MbUnit..http://www.mbunit.org

建立在 NUnit 之上。MbUnit 引入了很多更高层测试技术。这个包还包括集成组合测试、报表、行测试、数据驱动测试以及其他很多概念。

HTMLUnit ....http://htmlunit.sourceforge.net

在另一个自动化测试框架（如 JUnit）内部使用，HTMLUnit 会模拟一个 Web 浏览器来测试 Web 应用。

HTTPUnit...http://httpunit.sourceforge.net

HTTPUnit 非常类似于 HTMLUnit，不同的是，它使用 HTTP 请求和响应来完成测试。

JWebUnit .... http://jwebunit.sourceforge.net

JWebUnit 在 HTTPUnit 基础之上，提供了一个高层 API 实现 Web 应用导航。

可用的测试框架（测试工具）

Cobertura.. http://cobertura.sourceforge.net

Cobertura （西班牙语的 coverage）是一个代码覆盖率工具。运行一组测试时，它会告诉你测试了多少代码。

Clover.. http://www.cenqua.com/clover

这也是一个代码覆盖率工具，Clover 集成了面向大多数流行 IDE 的插件。

Fit...http://fit.c2.com

Fit 是一种基于表格的验收测试方法，很独特，对用户也很友好。即使不打算使用也值得你了解一下。

Fitnesse .... http://fitnesse.org

这是 Fit 的一个扩展。Fitnesse 既是一个独立的 Wiki，也是一个验收测试框架。

WinRunner ....http://www.mercury.com

WinRunner 是一个企业级工具，用于完成功能和回归测试（当然价格也不菲）。

LoadRunner ....http://www.mercury.com

与 WinRunner 出自同一家公司，LoadRunner 用来处理性能和压力测试。

Empirix E-Tester...http://www.empirix.com

Empirix 是一个 Web 记录／回放工具，内嵌在 IE 中。

Watir .... http://wtr.rubyforge.org

这个测试工具用来驱动 IE 中的自动测试。通过驱动 IE，它能解决模拟特定浏览器的 Web 页面解释问题。这个工具基于 Ruby，现在越来越流行。

Systir .... http://atomicobject.com/systir.page

这也是一个基于 Ruby 的工具，专为驱动采用其他语言的测试而设计。

关键概念

API 测试代码用来访问自动化测试框架的的编程接口。

测试创建方法（methodology for creating test） 自动化测试框架提供的框架。

用户界面（user Interface） 用户如何创建、保存、运行和维护测试。

测试引擎（test engine） 具体运行所创建测试的程序。

结果显示（result display） 如何得知测试是否通过。

如何选择

测试类型

这个工具或自动化测试框架是否可以运行你需要的测试（例如，功能测试、性能测试等）？

对所测试内容的支持

这个工具是否可以测试你的应用代码？能不能测试你的网站？

支持的编程语言

可以测试你本地使用的语言吗？还是必须学习一种新技术？

灵活性

你能创建并运行程序需要的那种测试吗？

开放格式

你的测试工具能与其他工具集成吗？

更多有关信息

见实践 7。

要更全面地了解测试框架，请访问以下网站：

- http://www.xprogramming.com/software.htm；
- http://www.testingfaqs.org/t-unit.html。

附录 H

建议阅读书目

这些书中有些我们自己读过，有些是审稿人建议的。请查看这个书目，试着挑出你还没有读过的一两本书读一读。

通用

《程序员修炼之道》（*The Pragmatic Programmer*），作者 Andy Hunt 和 Dave Thomas。这本书是有关个人实践的一本经典之作。每一个靠写代码为生的人都应该把它作为案头必备参考。

《精通正则表达式》（*Mastering Regular Expressions*），作者 Jeffrey Friedl。要处理我们遇到的文本，最强大的方法莫过于正则表达式，但是使用正则表达式时，就像是“龙来了”[①]，总是麻烦不断。这本书巧妙地为了我们扫除了这些障碍。

《人月神话》（*The Mythical Man-Month*），作者 Frederick Brooks。读过这本书之后（我甚至还是大学里读到的第一版，千真万确!），就会意识到软件开发远不只是编写一个程序那么简单。

《计算机程序设计艺术》（*The Art of Computer Programming*），作者 Donald Knuth。这套书有多卷，对经典计算机科学做了一个全面介绍。

《死亡之旅》（*Death March: The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*），作者 Edward Youdon。“死亡之旅”项目在软件行业很有名。应该对这些有所了解，以免被它们拖垮。

《重构：改善既有代码的设计》（*Refactoring: Improving the Design of Existing*

① 在中国文化中“龙”象征着“高贵，至高无上”，而在西方语言中，“dragon”的文化意象的寓意却截然相反，这里可以把“龙”理解为“麻烦”。——译者注

Code），作者 Martin Fowler。这本书很好地介绍了重构的基本思想，并给出了大量实用建议。

《重构与模式》（*Refactoring to Patterns*），作者 Joshua Kerievsky。你已经读过模式，也已经了解重构。现在来看看重构项目中的各种模式。

《企业集成模式》（*Enterprise Integration Patterns*），作者 Hophe 和 Woolf。这本书对于分布式 N 层系统的意义就相当于《设计模式》对于单个程序的价值。同样是让别人指出常见的陷阱和解决方案。读这本书会为你以后的项目提供一些绝好的想法。

《修改代码的艺术》（*Working Effectively with Legacy Code*），作者 Michael Feather。这是一本实用指南，会告诉你如何对继承的产品进行测试、重构和扩展。

《代码大全》（*Code Complete*），作者 Steve McConnell。这本书相当著名，汇集了构建软件的大量最佳实践。

Ruby

《Programming Ruby 中文版》（*Programming Ruby*），作者 David Thomas、Chad Fowler 和 Andy Hunt。这本关于 Ruby 的书主要针对我们这些不讲日语的人。要掌握这种语言，你想了解的一切都可以在这本书中找到。

The Ruby Way，作者 Hal Fulton。补充了《Programming Ruby》没有谈到的内容。这本书可以节省你的大量时间，学会如何充分使用这个语言完成一些常见任务。

Java

《Java 网络编程》（*Java Network Programming*），作者 Merlin Hughes 等。这本书会教我们如何编写网络代码。这就足够，不用再多说了。（Java 部分只是一个额外奖励。）

《项目自动化之道》（*Pragmatic Project Automation*），作者 Mike Clark。这是一本全面的指南，介绍了如何建立项目基础设施并实现自动化。对于开发人员来说，不论采用何种语言，这都是一本很不错的书！

《单元测试之道 Java 版》（*Pragmatic Unit Testing in Java*），作者 Andy Hunt

和 Dave Thomas。这是一本很好的 JUnit 入门指南（另外也有了面向 C#和.NET 的版本）。

软件方法

《测试驱动开发》（*Test Driven Development*），作者 Kent Beck。除了测试，这本书还包括大量设计方面的内容，它帮我们考虑了有关程序设计和实现的想法。

《Crystal Clear：小团队的敏捷开发方法》（*Crystal Clear: A Human-Powered Methodology for Small Teams*），作者 Allistair Cockburn。Crystal 是一种流行的敏捷方法。

《解析极限编程：拥抱变化》（*Extreme Programming Explained: Embrace Change*），作者 Kent Beck。对 XP 感兴趣？那就应该读读这本书。

《应用极限编程：积极求胜》（*Extreme Programming Applied: Playing to Win*），作者 Ken Auer。普遍认为这是所有 XP 书中最为实用的一本，这也是所有使用 XP 的人的又一本必读书。

《敏捷软件开发：使用 SCRUM 过程》（*Agile Software Development with Scrum*），作者 Ken Schwaber 和 Mike Beedle。Scrum 是一种非常好的轻量级方法。这个名字源自于橄榄球。

源代码管理

《版本控制之道：使用 CVS》（*Pragmatic Version Control Using CVS*），作者 Dave Thomas 和 Andy Hunt。尽管我们使用 CVS 已经很多年，但我们并不像自以为的那么了解它。这本书回答了大量问题（甚至有些问题我们都没有意识到）。

《版本控制之道：使用 Subversion》（*Pragmatic Version Control Using Subversion*），作者 Mike Mason。学习 Subversion 的人必读的一本书。

其他

The Little Schemer，作者 Daniel Friedman 和 Matthias Felleisen。这本书可以让我们拓展视野，从一种全新的角度来考虑程序、算法和递归。它讨论的是跳出主流开发模式思考问题！

《Dynamic HTML 权威指南》（*Dynamic HTML: The Definitive Reference*），作者 Danny Goodman。我们曾在一个漫长的夏天把它奉为跨浏览器 Web 开发的圣经。对于我们遇到的每一个问题，这本书几乎都有解释，而且通常还会提供解决方案。

Bugs in Writing: A Guide to Debugging Your Prose，作者 Lyn Dupre。如果你计划编写一些代码，应该在开始之前先要读这本书，而不是在编写代码的同时读（像我们一样！）。Lyn 诙谐而有效的建议可以帮助你把想法变成可以理解的书、文章和报告。

《UML 精粹》（*UML Distilled*），作者 Martin Fowler。并不是使用 UML 才能从这本书获益。这本书对统一建模语言（UML）做了很好的介绍，并指出如何以可视化方式表示软件。

领导力和关系

《领导力 21 法则》（*21 Irrefutable Laws of Leadership*），作者 John Maxwell。就像万有引力定律一样，你可以学习也可以违反这些定律。这本书对于人们如何领导（或者应该如何领导）很有见地。

《高效能人士的七个习惯》（*Seven Habits of Highly Effective People*），作者 Stephen Covey。这本书清楚地解释了如何组织你的生活，以及为什么要这么费心去做！只要你希望生活或工作得更有效，就应当读一读这本书。

《人件》（*Peopleware: Product Projects and Teams*），作者 DeMarco 和 Lister。这是一本经典的书，讨论了团队可能遇到的常见问题以及如何避免这些问题。另外，它还展示了一些重要的思想，你的团队可以利用这些思想避免重蹈很多其他团队的覆辙。

《人性的弱点》（*How to Win Friends and Influence People*），作者 Dale Carnegie。这本书是关于成功的关系（包括人际关系和公共关系）的一本权威指南。这本书售出了 1500 余万册，绝对是有道理的。如果你没有读过，真的应该读一读。

参考书目

[Cla04] Mike Clark. *Pragmatic Project Automation. How to Build, Deploy, and Monitor Java Applications*. The Pragmatic Programmers, LLC, Raleigh,

NC, and Dallas, TX, 2004.

[Coc01] Alistair Cockburn. *Agile Software Development*. Addison Wesley Longman, Reading, MA, 2001.

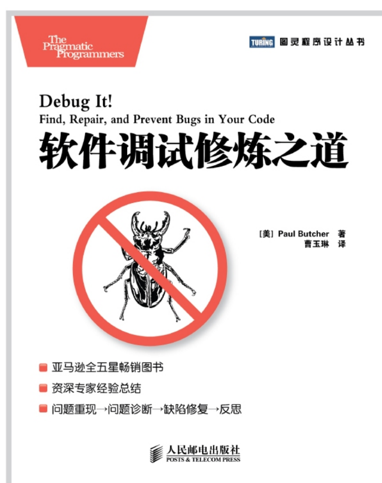
[Cus03] Cusumano. A global survey of software development practices. Technical Report 178, MIT Sloan School of Mgmt, June 2003. http://ebusiness.mit.edu/research/papers/178_Cusumano_Intl_Comp.pdf.

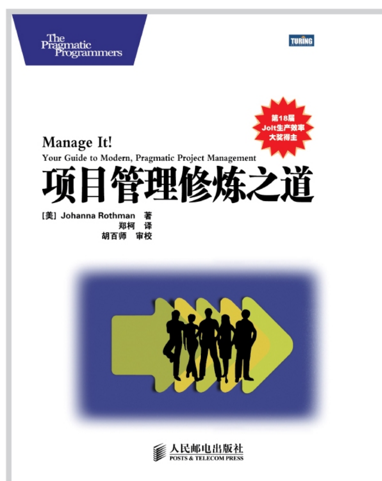
[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.

[TH03] David Thomas and Andrew Hunt. *Pragmatic Version Control Using CVS*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2003.

[You99] Edward Yourdon. *Death March: The Complete Software Developer's Guide to Surviving'Mission Impossible'Projects*. Prentice Hall, Englewood Cliffs, NJ, 1999.

[Zei01] Alan Zeichick. Debuggers, source control keys to quality. *Software Development Times*, March 2001.


The Pragmatic Programmers
TURING 图灵程序设计丛书
Debug It!
Find, Repair, and Prevent Bugs in Your Code
软件调试修炼之道
[美] Paul Butcher 著
曹玉琳 译
亚马逊全五星畅销图书
资深专家经验总结
问题重现→问题诊断→缺陷修复→反思
人民邮电出版社
POSTS & TELECOM PRESS


The Pragmatic Programmers
TURING
第18届 Jolt生产效率大奖得主
Manage It!
Your Guide to Modern, Pragmatic Project Management
项目管理修炼之道
[美] Johanna Rothman 著
郑柯 译
胡百师 审校
人民邮电出版社
POSTS & TELECOM PRESS


The Pragmatic Programmers
TURING 图灵程序设计丛书
Practices of an Agile Developer
高效程序员的45个习惯
敏捷开发修炼之道
[美] Venkat Subramaniam Andy Hunt 著
钱安川 郑柯 译
人民邮电出版社
POSTS & TELECOM PRESS

“我想大多数人都应该读这本书。尽管它主要是写给技术主管的，但我觉得所有开发人员都能从中受益。只要掌握了本书的思想，我相信项目经理们都能获益匪浅。阅读本书能让人身心愉悦，我深深地沉醉其中。本书的内容框架和组织结构更是让人印象深刻。极力推荐！”

——技术作家Harry Newton，news@UK

“如果你只满足于做一个平庸的开发人员，那么这本书绝对不适合你！”

——亚马逊读者评论

Ship it! A Practical Guide to Successful Software Projects

软件项目成功之道

如何成功地开发并交付软件是许多软件企业面临的难题。本书由专业开发人员编写，汇集大量实用性建议及成功的项目团队需要用到的工具和技术，通过42个技巧全面概括了软件开发的方方面面，为开发人员指明了最适用的方法。本书还给出了大量软件开发实践，供读者认真思考并将其中涉及的具体概念转化为自己的所得。

就像其他Pragmatic图书一样，作者不仅分享了自己多年的软件开发经验，而且将枯燥乏味的软件开发过程描写得趣味横生，相信本书会对你的软件研发工作有所启迪，从而改变一些思维定式和研发模式。

- 根除项目延期之痼疾
- 打开软件项目成功之门
- 经验宝贵，篇幅精练，讲解透彻

The Pragmatic Programmers

图灵网站：www.turingbook.com　热线：(010)51095186转604
反馈/投稿/推荐信箱：contact@turingbook.com
有奖勘误：debug@turingbook.com

分类建议　计算机/软件工程

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-25965-3
定价：39.00元